UltraFast 高层次生产力设计方法指南

UG1197 (v2018.2) 2018 年 6 月 6 日

条款中英文版本如有歧义，概以英文文本为准。

XILINX

修订历史

下表列出了本文档的修订历史。

章节	修订总结
2018 年 6 月 6 日 2018.2 版	
“Vivado HLS 设计流程”	更新 Vivado HLS 设计流程。

目次

XILINX

第 1 章

高层次生产力设计方法指南

关于本指南

赛灵思可编程器件含有数百万个逻辑单元 (LC)，集成了当前越来越多的复杂电子系统。本高层次生产力设计方法提供了在短设计周期内开发此类复杂系统的一套最佳做法。

这种方法以下列概念为重点：

- 对宝贵的差异化逻辑使用并行开发流程，实现您的产品在市场上的差异化，且 shell 可用于将 IP 与生态系统的其它部分集成。
- 广泛使用基于 C 语言的 IP 开发流程开发差异化逻辑，让仿真速度相对于 RTL 仿真成倍增长，并且能提供时序准确和得到优化的 RTL。
- 使用现有的预验证、块和组件级 IP 来快速构建 shell，将差异逻辑封装在系统中。
- 使用脚本，针对从准确设计验证直至编程 FPGA 的流程实现高度自动化。

本指南中的建议是过去多年的广泛专家级用户的经验总结。与传统的 RTL 设计方法相比，它们不断提供了下列改进：

- 设计开发时间加快 4 倍。
- 衍生设计开发时间加快 10 倍。
- 结果质量 (QoR) 提高 0.7 倍到 1.2 倍。

虽然本指南以大型复杂设计为重点，讨论的实践也适用于且已被成功地应用到各种类型的设计中，包括：

- 数字信号处理：
  - 图像处理
  - 视频
  - 雷达
- 汽车
- 处理器加速
- 无线
- 存储
- 控制系统

对新设计方法的需求

在当今日益复杂的电子产品中使用的先进设计正在挑战器件密度、性能和功耗的极限，同时也使设计团队面临挑战，要求他们必须在限定的预算内按时完成设计目标，获得机会窗口。

解决这些设计挑战的高效方法是把更多时间用于较高层次的描述，从而获得最快的验证时间和最大的生产力提升。

对新设计方法的需求在下图中得到充分体现。每个区域的面积代表设计流程中每个阶段的开发工作量的比例。

- 对传统 RTL 方法而言，大部分工作耗费在细节的实施工作上。
- 在高层次生产力设计方法中，大部分工作用于设计和验证您是否构建了正确的系统。

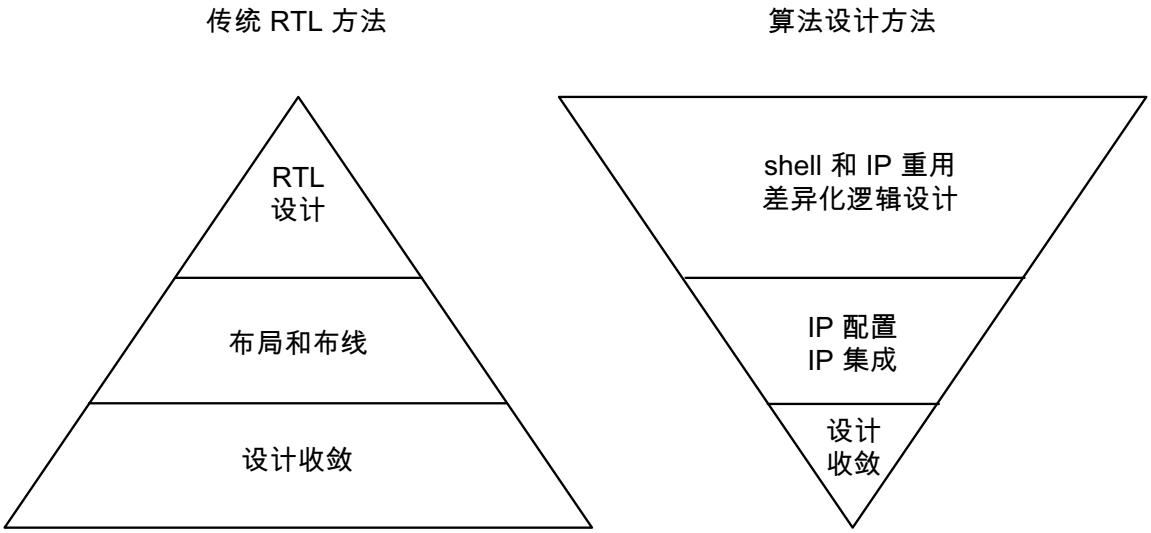

图 1-1：高层次生产力设计方法比较

传统方法

传统设计开发首先是由有经验的设计人员估计如何用新技术实现自己的设计，完成寄存器传输级 (RTL) 的设计采集，通过综合和布局布线执行一些尝试，确认自己的估计，然后继续开展其余部分的设计采集工作。一般完成这项工作的方法是逐次综合每个块，以重复确认设计实现细节可接受。

确认设计能提供所需功能的主要方法是仿真该 RTL。尽管 RTL 描述方式具备位准确和周期准确的性质，但这种高度准确性也使得仿真速度过慢且易出错误。

只有当设计中的所有块都已经采集到 RTL 中才能够对系统开展完整验证，往往会造成对 RTL 的调整。在系统中的全部块验证完毕后，就可以集中布局布线，早期对时序和占位面积的估算准确性要么完全相符，要么会发现不准确的地方。这也往往会导致对 RTL 的修改，重新启动系统的又一次验证和又一次再实现。

设计人员现在往往需要在给定项目中实现数十万行 RTL 代码，把大部分时间花在细节的实现工作上。如图 1-1 中所体现，设计人员把更多时间花在实现设计上，而不是设计所有产品保持竞争力所必须的新颖创新的解决方案。

无论是采用更新的技术以提升性能，还是采用更缓慢的技术以提供更具竞争力的定价，都意味着大部分 RTL 必须重新写入。设计人员必须重新实现寄存器间的大量逻辑。

高层次生产力设计方法指南

高层次生产力设计方法沿袭了较为传统的 RTL 方法的基本步骤，如图 1-1 所示。但是，它能够让设计人员把更多时间花在设计增值解决方案上。高生产力方法的主要属性有：

- shell 概念，即把 I/O 外设和接口采集到独立的设计项目中，与差异化逻辑并行开发和验证。
- 使用基于 C 语言的 IP 仿真，让仿真速度与传统 RTL 仿真相比减少到数量级，为设计人员提供了设计理想解决方案的时间。
- 运用赛灵思 Vivado® Design Suite，使用基于 C 语言的 IP 开发、 IP 重复使用和标准接口实现时序收敛的高度自动化。
    - 使用 Vivado IP 目录方便地重复使用您自己的块和组件级 IP，还能方便地获取已通过验证且已知能在该技术中良好实现的赛灵思 IP。

高层次生产力设计方法中的所有步骤都能交互式地执行，或使用命令行脚本执行。所有手工交互的结果都可以保存到脚本，实现从设计仿真直至 FPGA 编程的整个流程的完全自动化。根据您的设计和 RTL 系统级仿真的运行时间，该流程可在电路板上生成 FPGA 比特流并测试设计，一般能在任何 RTL 设计仿真完成之前开展。

创建衍生设计时，还将得到更加明显的生产力提升。就像修改工具选项一样简单，基于 C 语言的 IP 与不同的器件、技术和时钟速度可轻松对应。完全脚本化的流程加上通过 C 语言综合实现的自动时序收敛，意味着能够迅速地完成衍生设计的验证和组合。

设计进程

下图显示了设计进程的各个步骤。

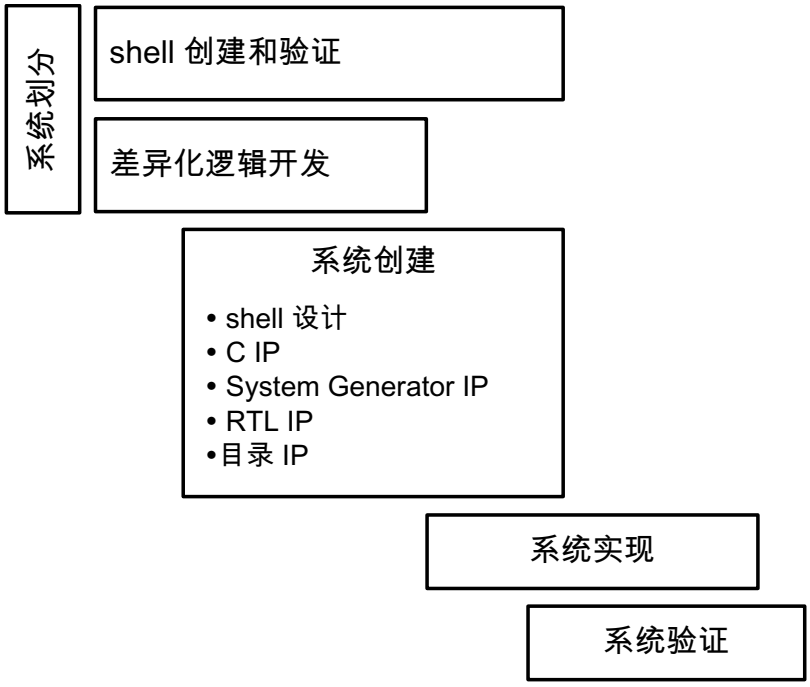

图 1-2：高层次生产力设计流程

该设计流程中，第 2 章"系统设计"中描述的系统分区这一初始阶段后，几个后续步骤可以并行执行。

- shell 开发流程：通过使用 Vivado IP 集成器和 IP 目录， Vivado Design Suite 能实现快速高效的块级集成。系统性能关键方面的很大一部分，包括详细接口创建、验证和管脚分配，可以独立到一个并行开发项目中，从而重点关注。该流程详见第 3 章"shell 开发"。
- 基于 C 语言的 IP 开发：使用 RTL 仿真（取决于设计、主机等条件）完整的一帧视频需要大约一到两天时间。使用 C/C++ 执行同样比特级精度仿真只需大约 10 秒钟。基于 C 语言的开发流程带来的生产力改善不容忽视。该流程详见第 4 章"基于 C 语言的 IP 开发"。
- 系统创建：运用 Vivado IP 集成器和 IP 目录，使用 shell 设计、原有 RTL IP、System Generator IP 和赛灵思 IP 就可以把基于 C 语言的 IP 迅速结合到系统块设计中。自动化接口连接功能和系统创建的脚本化功能意味着系统在整个 IP 开发流程中能够迅速地反复生成。该流程详见第 5 章"系统集成"。
- 系统实现：使用经过验证的 shell 设计、自动为器件和时钟频率优化的基于 C 语言的 IP、现有的经验证的 IP，并使用业界标准的符合 Arm AMBA® AXI4 协议的接口把它们全部连接起来，您就可以最大程度地节省花在设计收敛上的时间。只需单击几次鼠标或是使用脚本化流程，就可以从系统块设计启动这一流程。该流程详见第 5 章"系统集成"。
- 系统验证：系统验证可以使用门级精度的 RTL 仿真和/或通过编程 FPGA 并在电路板上验证设计。由于 RTL 仿真用于验证系统，而非开发过程中用于验证和设计的迭代性仿真，故在设计流程结束时只需要一次仿真。该流程详见第 5 章"系统集成"。

访问技术文档和培训资料

在适当的时间获得正确的信息，对于及时设计收敛并确保整体设计成功而言十分重要。参考手册、用户指南、教程和视频能够帮助您尽快掌握 Vivado Design Suite。本节为您列出了部分技术文档和培训资料的来源。

使用 Documentation Navigator

Vivado Design Suite 配套提供赛灵思 Documentation Navigator （图 1-3），用于访问和管理全套赛灵思软/硬件文档、培训资料和辅助材料。借助 Documentation Navigator，您可查看赛灵思最新及过去的技术文档。您可通过版本、文档类型或设计任务来过滤技术文档显示内容。结合搜索功能可帮助您快速找到正确的信息。 “Methodology Guides”是技术“Document Types”下的过滤器之一，借助该过滤器，您几乎可以在瞬间找到任何的方法指南。

赛灵思通过 Documentation Navigator，使用“Update Catalog”功能，为您提供最新的技术文档。该功能可提醒您有可用的目录更新内容，并提供有关文档的具体信息。赛灵思建议您在出现提醒时要更新目录，以使其保持最新。此外，您可以为指定的文档建立本地技术文档目录并对其进行管理。

Documentation Navigator 中有一个“Design Hub View”标签。“Design Hub”是指与设计活动（如应用设计约束、综合、实现，以及编程和调试等）相关的文档集。文档和视频被纳入每个设计中心内，以简化相关领域的学习过程。每个设计中心均包含“Getting Started” （快速入门）部分、 “Support Resources” （辅助性资料）部分 （包含该流程的 FAQ），以及“Additional Learning Material” （更多学习资料）。 “Getting Started”部分可为新用户提供清晰的入门指导。对已经熟悉该流程的用户来说， “Key Concept”和“FAQ”部分可能是他们比较感兴趣的内容，有助于他们获得 Vivado Design Suite 相关专业知识。

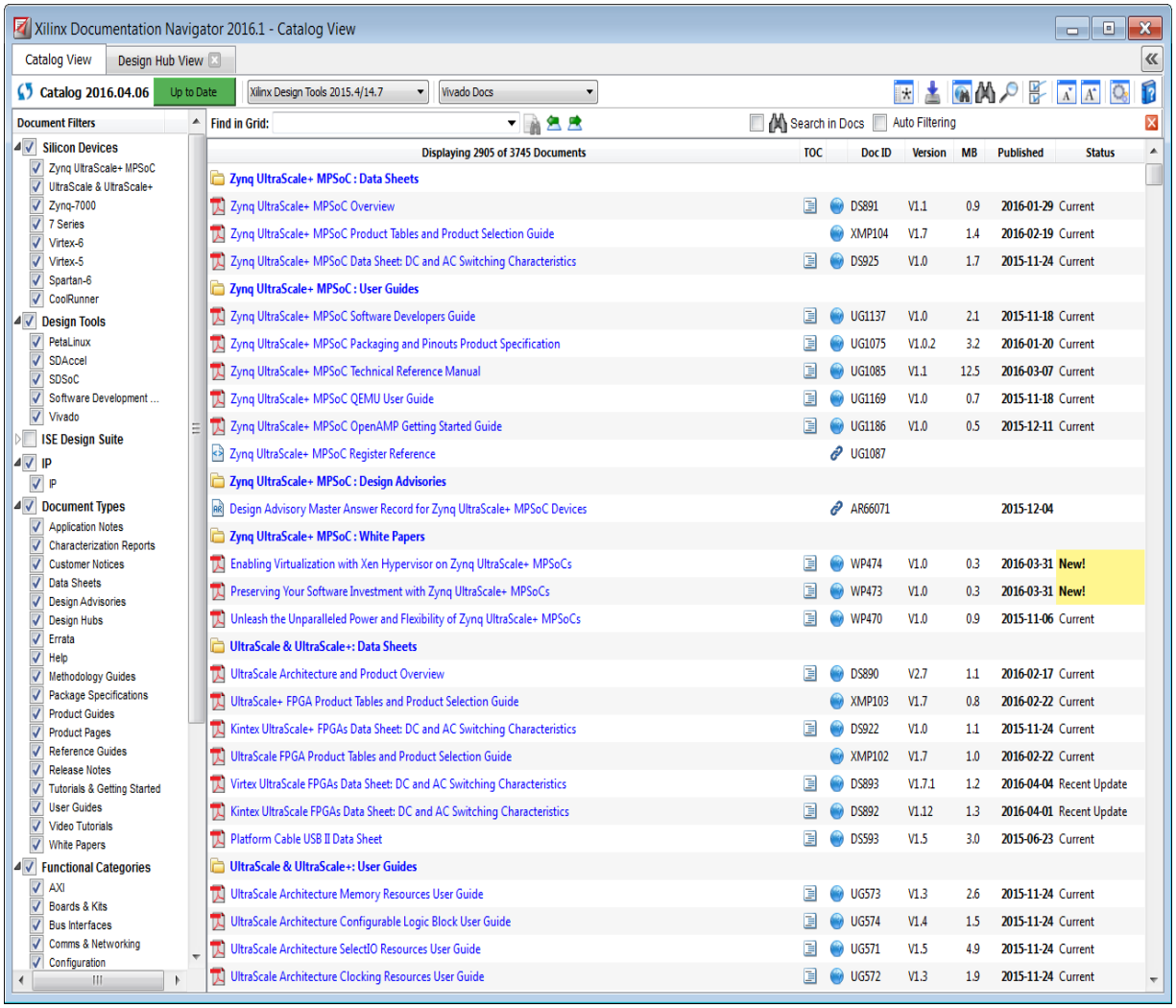

图 1-3：赛灵思 Documentation Navigator

XILINX

第 2 章

系统设计

简介

在您开始项目之前，一个重要前提是需要对系统的设计和组合方法有清晰的理解。在任何复杂的系统中都存在通向解决方案的多条路径。这些路径由您的选择而定，包括创建什么样的完整 IP 块、重复使用哪些 IP 块、使用哪些工具和方法验证 IP/集成 IP 到系统中以及使用什么工具和方法检验系统。

本章的目的是探讨您做出的系统分区选择和回顾 Vivado® Design Suite 中有助于系统开发流程自动化的关键特性。

- “系统分区”
- “系统开发”

系统分区

在典型设计中，位于设计边缘处的逻辑专门用于与外部器件连接，一般使用标准接口。这方面的实例有 DDR、千兆位以太网、PCIe、HDMI、ADC/DAC 和 Aurora 接口。对同一家公司内的多种 FPGA 设计而言，这些接口和用于实现它们的组件一般是标准的。

在高层次生产力设计方法中，该逻辑与核差异化逻辑彼此独立，被视为 shell。下图所示的即为 shell 块设计示例。下图中心的阴影部分指出了可以添加差异化逻辑或 shell 验证 IP 的区域。

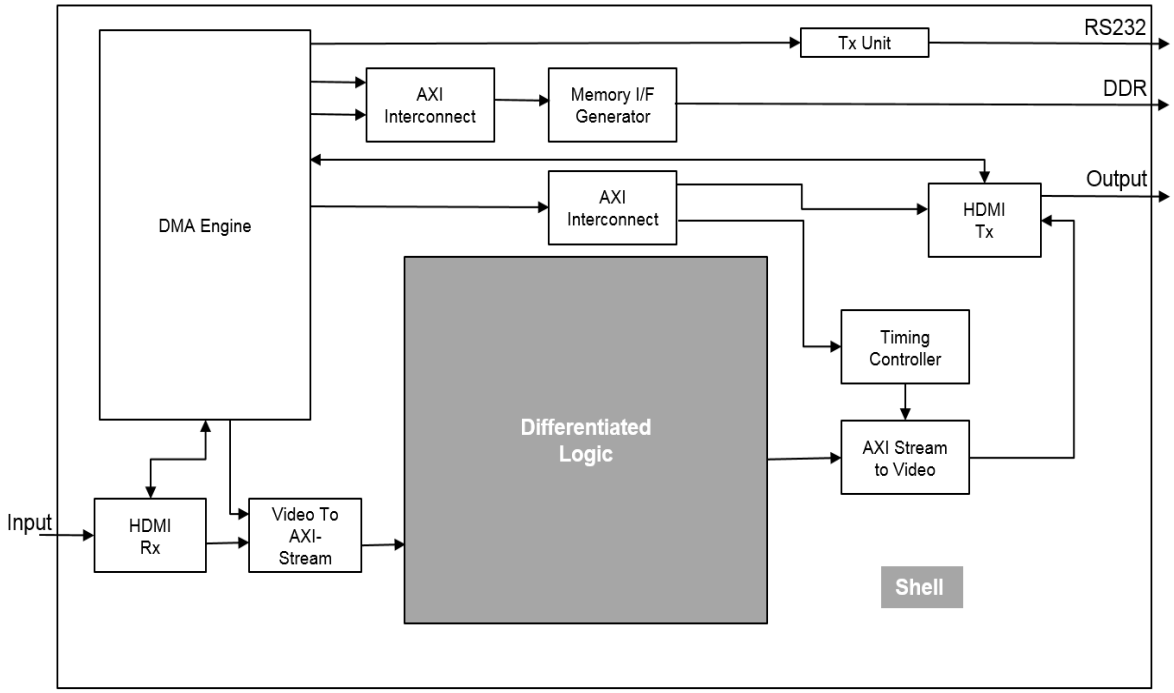

图 2-1：shell 设计示例

这种方法的主要优势有：

- shell 的开发和验证独立于设计的其余部分。
- 电路板级集成和器件管脚分配由并行开展工作的独立专门团队负责处理。
- 可保存和重复使用 （甚至重新编辑）该 shell，便于迅速实现多种衍生设计。
- 差异化逻辑的开发和验证独立于 shell。
- 预验证的 shell 和差异化逻辑被迅速集成到一个完整系统中。

在分区您的系统时，第一项任务就是判断什么要实现在 shell 中，什么要作为差异化逻辑实现。

shell 设计

Shell 设计为高生产力方法提供了两项关键属性：

- 将标准接口逻辑从差异化逻辑分开，可以让两者的开发和验证并行开发。
- 创建能够用于迅速开发衍生设计的可重复使用的设计或 shell。理想情况下 shell 应包含设计的标准组成部分，例如设计接口和接口 IP。不过 shell 也可以包含用于预处理或后处理的块。如果处理功能独立于核设计 IP 且处理功能可在多种设计中使用，更理想的方法是把这些块布局在 shell 内。这种平台重复使用方法便于从 shell 上方便地移除块。

无论您决定纳入 shell 设计的逻辑是什么，该 shell 设计的关键属性之一是内部接口，即与内部设计 IP 连接的接口应使用标准接口实现。使用 AXI 等标准内部接口能带来下列特点，能增强 shell 的可重复使用性：

- 便于 shell 与尚待开发的设计 IP 方便地连接
- 确保在验证 shell 的同时也完成对内部接口的验证

- 能够使用“IP 集成器与标准接口”介绍的高生产力集成功能

即便您最初只考虑一个设计，基于平台的方法让您能够在初始设计实现后轻松地创建衍生设计。

关于 shell 开发开发和验证的更详细说明请参见第 3 章"shell 开发"。

IP 设计

IP 开发流程的主要特性是它只包含能够区分产品与 shell 的 IP。

该设计 IP 非标准 IP，需要开发。大部分开发工作用于运行仿真，以验证设计能否提供正确的功能。通过排除不会给处于开发中的新功能造成影响的标准块，能最大程度地降低这一工作量和缩短仿真运行时间。这些标准块应处于 shell 内。

下图展示了一个将设计 IP 添加到 shell 设计的完整系统演示。完成后的系统的关键特性之一在于它可以包含不同来源开发的 IP，例如：

- 使用 Vivado HLS 由 C/C++ 生成的 IP
- 使用 System Generator 生成的 IP
- 使用 RTL 生成的 IP
- 赛灵思 IP
- 第三方 IP

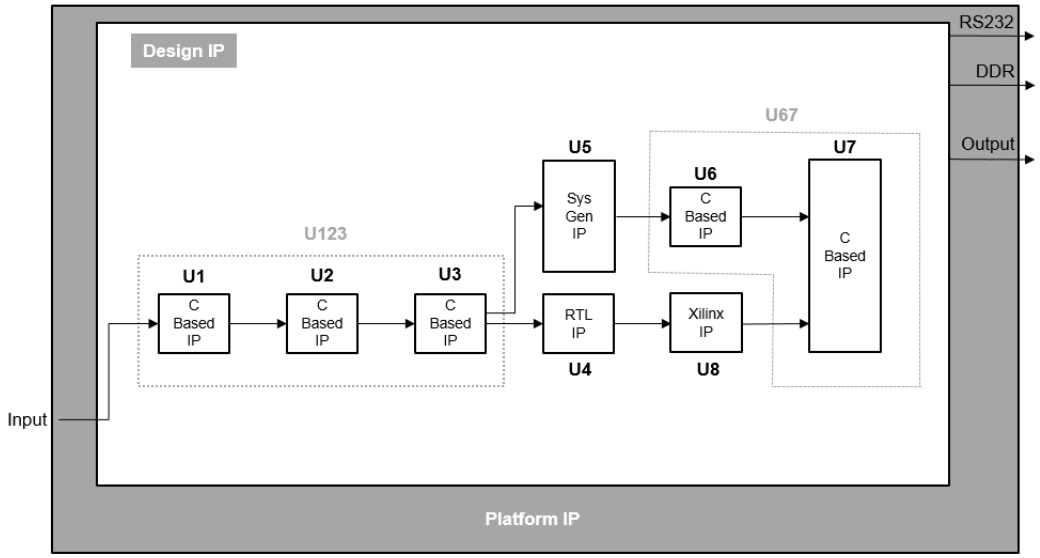

图 2-2：系统设计示例

在高生产力设计方法中，最显著的优势之一来自于 C 语言仿真的验证速度。从设计创建的角度来看，通过在开发过程中集中仿真 C 语言块能够带来明显的生产力改善。

- 高速 C 语言仿真便于设计人员迅速开发和验证准确的解决方案。
- 同时仿真多个 C 语言块有助于彼此验证各自的输出。
- 如果把多个 C 语言 IP 结合到一个 C 语言仿真中能够产生更明显的总体生产力优势。

图 2-2 突出表现了您在使用 C 语言 IP 时可能遇到的两难局面。块 U1、U2 和 U3 是都属于 C 语言 IP，它们可以被组合到单个顶层 U123 中。与此类似，块 U6 和 U7 是可以组合到单个 IP 块 U67 中的 C 语言 IP。您可以选择下列两种方法之一：

- 创建多个较小的 C 语言 IP 块，例如 U1、 U2、 U3、 U6 和 U7。
- 创建几个大型 C 语言 IP 块，例如上图中列出的 U123 和 U67。

从设计集成的角度出发，这两种方法之间并无区别。如果 IP 块是使用 AXI 接口创建的，使用 IP 集成器就能够方便地把它们集成到一起。对刚刚接触基于 C 语言 IP 开发的设计人员来说，较为理智的做法可能是使用较小块开展工作、学习如何独立优化每个小块、然后把多个小型 IP 块集成到一起。对已经熟悉 C 语言 IP 开发的设计人员而言，较为方便的做法是生成几个大型 C 语言 IP 块。

重要提示：关键的生产力优势在于能够在开发过程中把尽量多的 C 语言 IP 块集中到一个 C 语言仿真中进行仿真。

在上述情况下，用于验证块 U1、U2 和 U3 的 C 语言测试平台同样用于验证 U123。IP 生成的区别在于您或是在 Vivado HLS 中把 C 语言综合的顶层设置为 U123 功能，或是设置为 U1 功能，随后还有 U2 和 U3。

不管采用什么方法创建这些 IP 块，每个 IP 块都需要按下列方法进行独立验证：

- 用 C/C++ 开发的 IP 使用 Vivado HLS 的 C/RTL 协同仿真功能进行验证，方便采用用于验证基于 C 语言 IP 的同一 C 语言测试平台验证 RTL。
- 用 System Generator 开发的 IP 采用 System Generator 中提供的 MathWorks Simulink 设计环境进行验证。Simulink 环境通过使用预先定义的仿真元，能方便地生成复杂的输入激励物并分析复杂的结果。用 C/C++ 以及通过传统 RTL 生成的 IP 都能导入到系 System Generator 环境中，发挥这一验证功能的作用。
- 对用 RTL 生成的 IP，用户必须创建 RTL 测试平台来验证该 IP。
- 赛灵思和第三方供应商提供的 IP 已预先验证，不过您可能希望根据自己的配置参数集合创建测试平台来验证其运行。

在 IP 上使用标准 AXI 接口能实现 IP 彼此之间以及 IP 与 shell 设计的迅速集成。

系统开发

虽然使用 shell 和多个 IP 块对 FPGA 设计人员而言并非新概念，这种方法一般需要开发和仿真大量 RTL，多次整合数百乃至数千独立信号以完成下列连接：

- shell 到验证 IP
- shell 到核设计 IP
- shell 到衍生核设计 IP

鉴于在传统 RTL 设计进程中使用这种方法会因设计和验证工作产生大量额外的工时 （而且如果是在文本编辑器中进行，还容易发生错误），设计团队一般选择设计和集成所有内容。

Vivado IP 集成器能让这种方法成为可行，无需传统的 RTL 文件手工编辑即可迅速完成 IP 集成工作。

使用这一方法具有下列关键特性：

- Vivado IP 目录
- IP 集成器与标准接口

Vivado IP 目录

Vivado IP 目录是任何使用 IP 和重复使用 IP 的方法的基干。图 2-3 展示了有关高层次生产力设计方法的设计进程的另一种观点，主要展示了使用 IP 目录的位置和时间。

重要提示：使用 IP 目录是实现高层次生产力设计方法的关键。

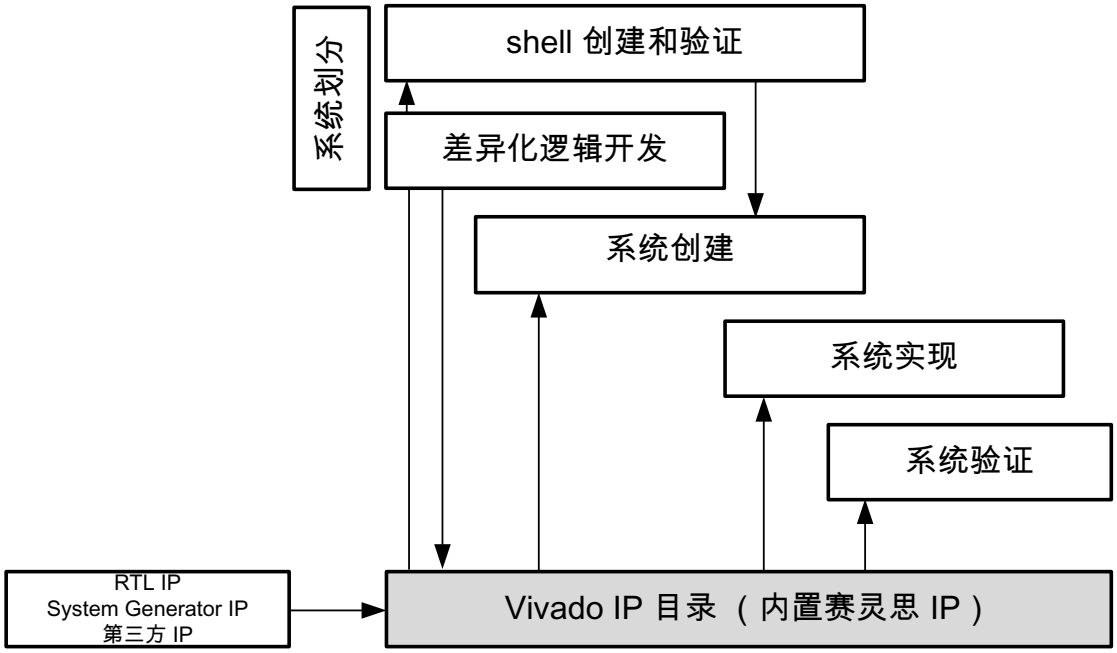

图 2-3：IP 目录与设计进程

IP 目录具有下列特性：

- 内含大约 200 个由赛灵思提供的 IP。如需了解更多信息，请参阅赛灵思 IP 页面 [参照 12]。
- 保存来自基于 C 语言的 IP 开发的输出。
- 能使用 System Generator、原有 RTL 和赛灵思合作伙伴 IP 加以强化。
- 内置大量接口 IP，支持使用原有 RTL Ip，在创建 shell 时广泛使用。
- 是系统集成过程中所有 IP 块的来源。
- 在系统集成和验证过程中提供 RTL 实现功能。

在 shell 开发过程中该 shell 可使用 IP 目录提供的 IP 在 IP 集成器中组合。其中可包括赛灵思提供的接口 IP （以太网、VGA、CPRI、串行收发器等）、赛灵思合作伙伴提供的 IP、供 IP 目录使用的作为 IP 的原有 RTL 封装或是 Vivado HLS 和 System Generator 创建的 IP。

关于把原有 RTL 封装为 IP 的详情，请参阅 《Vivado Design Suite 教程：创建和封装定制 IP》(UG1119) [参照 5]。

关于使用 System Generator 提供的 AXI 接口创建 IP 的详情，请参阅 《Vivado Design Suite 用户指南：使用 System Generator 开展基于模型的 DSP 设计》(UG897) [参照 6]。

Vivado HLS 的默认输出是用于 IP 目录的经封装 IP。详见“IP 封装”。

IP 集成器与标准接口

使用 Vivado IP 集成器可以迅速把 IP 块添加到菜单中并完成连接，是高生产力设计方法的关键使能手段。

重要提示：使用 Vivado IP 集成器实现高生产力的关键在于使用标准接口。

图 2-4 所示的是 IP 集成器内采集的块设计示例。

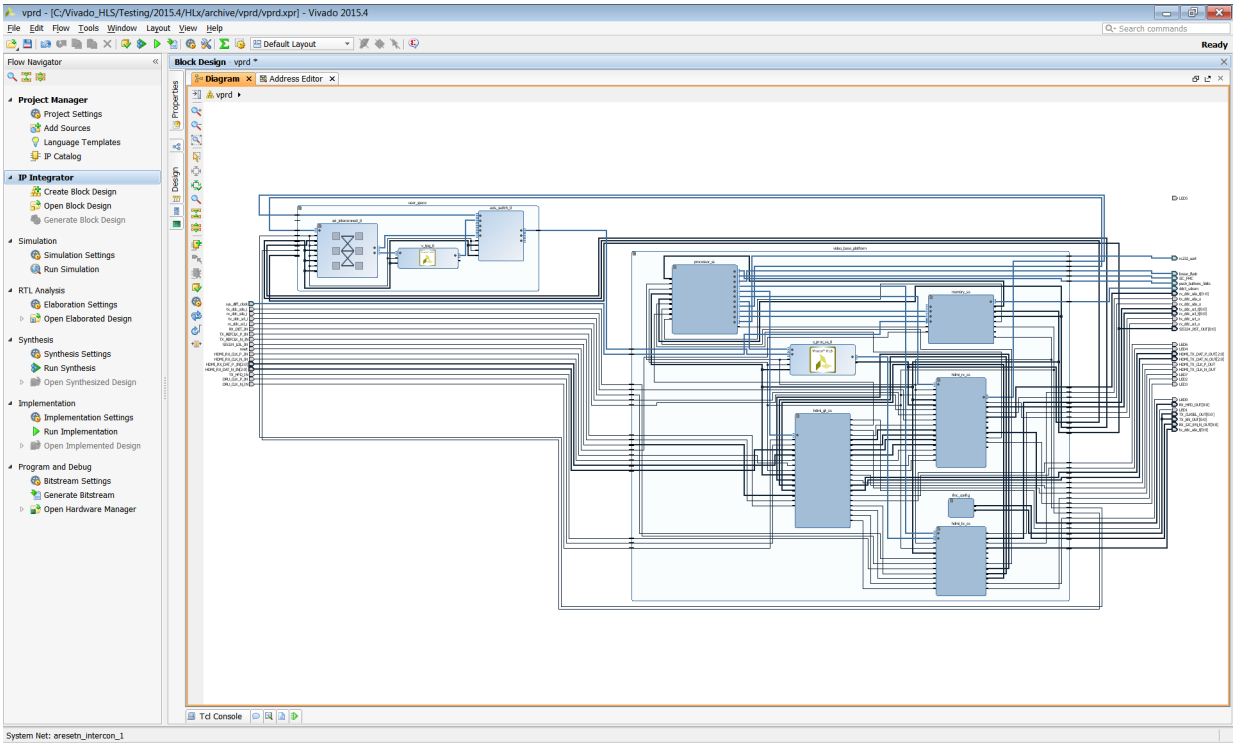

图 2-4：IP 集成器块设计

连接类型包括：

- 引脚级连接，比如时钟和复位信号。
- 总线级连接，比如 AXI、 AXI4-Lite 和 AXI4-Stream 总线。
- 电路板级连接，比如 DDR。

IP 集成器中的连接通过使用鼠标以图形方式把每个 IP 上的引脚连接起来。除了支持基本的比特级连接，还支持总线级连接并提供设计辅助功能。

下图主要体现的是总线级连接的优势。在本示例中，两个 AXI 主控制器接口连接在一起。注意只要第一个端口的连接完成，图上将用绿色记号标识出所有可能的有效连接。

重要提示：IP 集成器禁止非法连接。这避免了在使用手工编辑执行该流程的过程中发生的典型连接错误。

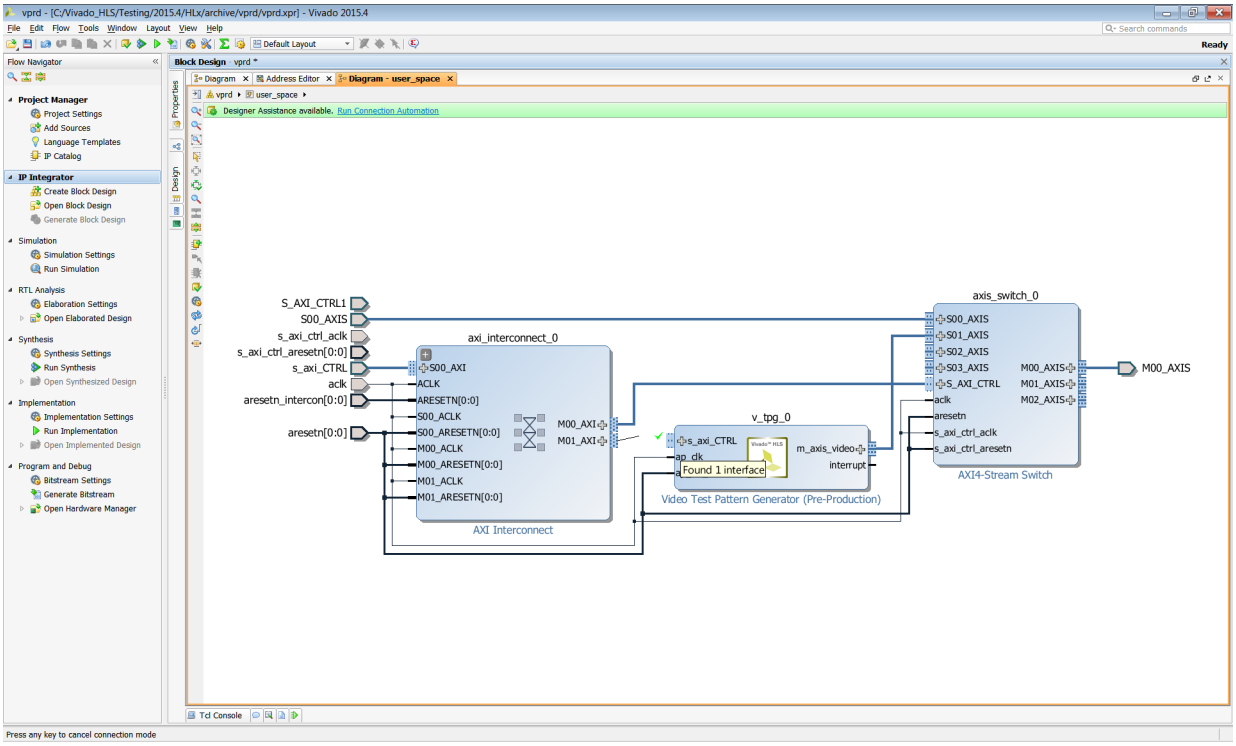

图 2-5：自动连接

还有一项通过使用标准 AXI 接口和 IP 集成器提供的生产力特性，那就是自动生成 AXI 互联 IP。图 2-6 展示了连接结果：

- 一个块上的 AXI 输出
- 另一个块上的 AXI4-Stream 输入

IP 集成器自动添加 AXI 互联 IP，以连接主控制器类型 IP 到流类型接口。

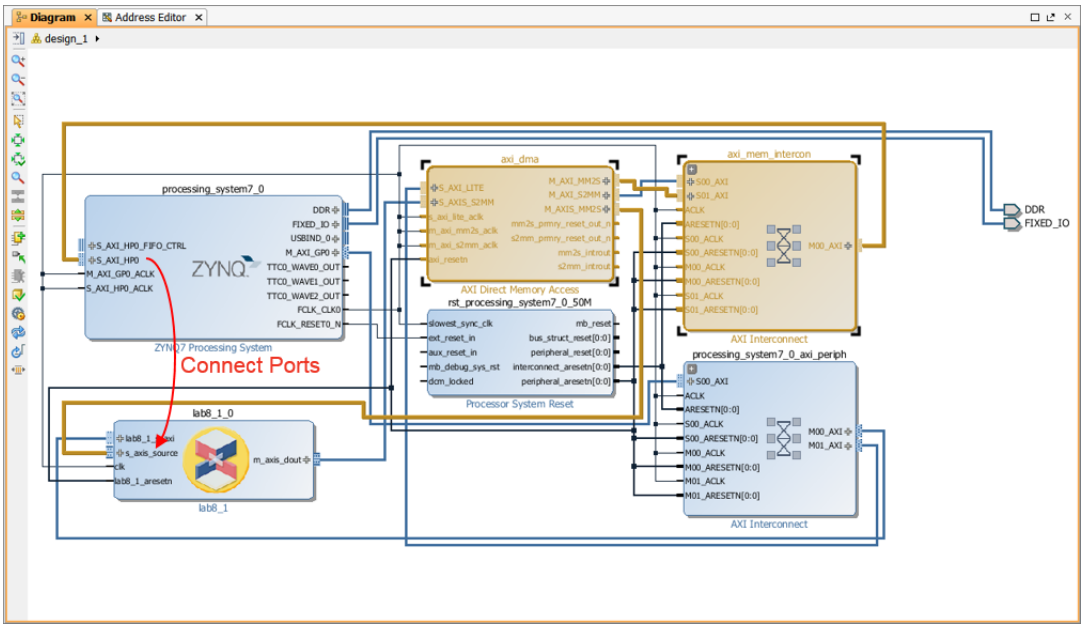

图 2-6：自动添加 AXI 互联 IP

IP 目录中提供了该 AXI 互联 IP，可以手工添加，不过 IP 集成器能自动完成这项任务。此外，如果把最终块设计保存为脚本， Tcl 命令就会指出需要连接到哪个引脚。

提示：当升级到新版本的 Vivado Design Suite 和赛灵思 IP 时，应重新运行脚本，从而确保其使用的是最新的互联逻辑。

在您的设计中使用标准接口的最后一种情况是为电路板级连接提供的设计辅助功能。除了可以选择目标器件， Vivado Design Suite 还可以选择目标板。 IP 集成器具有电路板级感知能力，能够自动完成电路板级连接。

在得到设计人员确认后， IP 集成器能自动完成 IP 和 FPGA 引脚之间的连接 （电路板连接）。

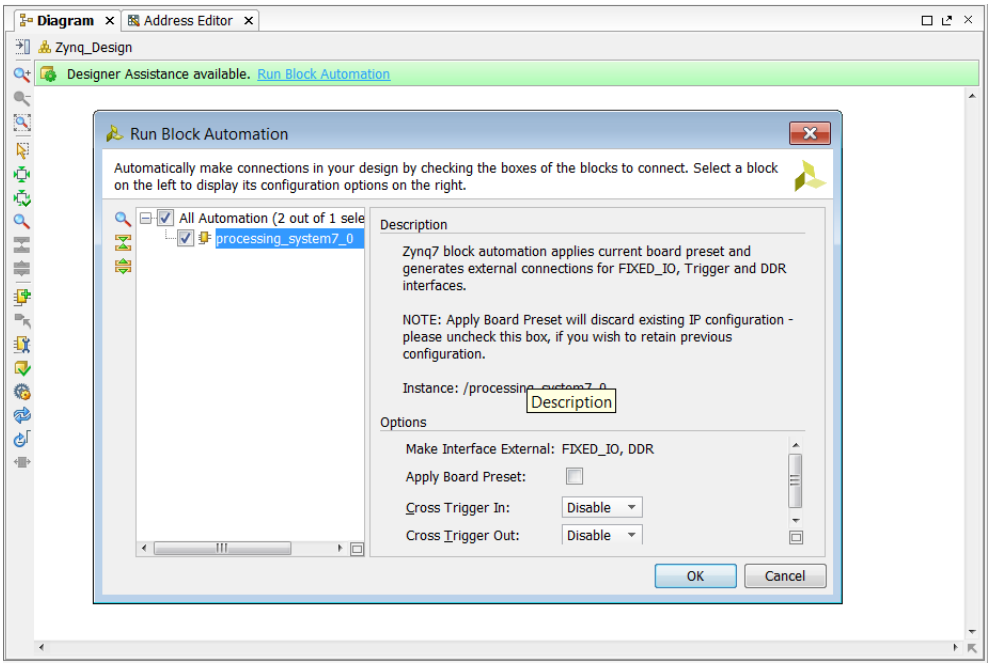

图 2-7：块自动化

IP 集成器能自动集成 IP 到块级原理图。其他特性包括使用验证功能的设计规则检查和为 AXI 互联 IP 自动添加时钟和复位逻辑。发挥这一自动化功能的优势，落实成效显著的 shell 方法的关键在于对片上通信使用标准接口和 AXI 接口。

XILINX

第 3 章

shell 开发

简介

高生产力设计方法提供的生产力优势关键在于使用 shell。一个 shell 设计包含所有的标准接口和处理块，提供核设计 IP 与系统其余部分的连接，并与核设计 IP 并行开发。

这种囊括 shell 设计的方法可提供重要的生产力提升，例如：

- 它能够实现独立于核设计进行的设计接口和 I/O 管脚分配的开发。
- 该方法可在核设计 IP 就绪之前开展设计接口的验证。
- 由于设计更小，缩短了接口的验证时间；它并不包含一般情况下占据大部分系统逻辑的核设计 IP。
- 这推进了一套生产力显著的设计重复使用方法，实现轻松地创建衍生设计。

shell 设计方法的简介如图 3-1 所示。该方法的一个重要特点是 shell 设计的重复使用。

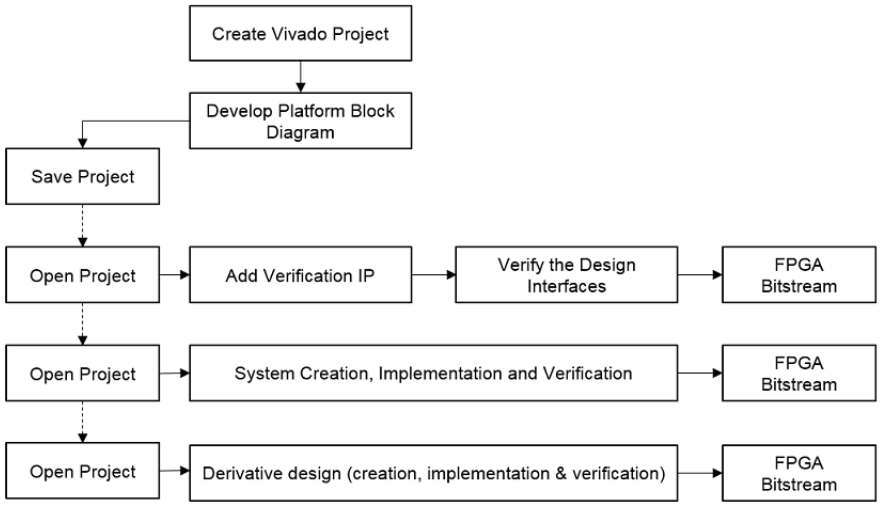

图 3-1：shell 方法

shell 开发包含两个同样重要的过程：shell 设计和 shell 验证。

shell 设计

shell 设计仅包括设计边缘，如上图所示，而且必须采用易于重新利用设计的形式。 shell 将保存并重新打开以构成多个项目的基础。

为实现如上图所示的流程从而达到设计重新使用所需级别， shell 设计应作为能够轻松保存和重新开通的 IP 集成器的块设计，以便构建其它设计项目的基础。

组装现有 IP

在 IP 集成器中使用 IP 目录提供的 IP，以块设计的形式组装 shell 设计。

重要提示：作为创建 shell 的的准备，请封装现有 RTL 或任何您希望在 shell 设计中使用的具体公司 IP，以备 IP 目录使用。这样您就可以将 IP 添加到 shell 块设计中。

如需了解有关如何为 IP 目录封装块的详情，请参阅 《Vivado® Design Suite 教程：创建和封装定制 IP》(UG1119) [参照 5]。

shell 设计项目

组装 IP 后，创建一个 Vivado RTL 项目。

培训：Vivado Design Suite QuickTake 视频：创建不同类型的项目详细介绍了 Vivado RTL 项目创建。

创建 Vivado 项目时：

- 将项目定义为一个 RTL 项目并选择“Do not specify any sources at this time”。 shell 设计源就是您封装在 IP 目录中的 IP。
- 理想情况下，选择该目标板作为赛灵思开发板。赛灵思开发板所用器件的 I/O 已完成配置。这样，在您已开发自己的定制板时能以最短时间开展工作，您也可以利用 IP 集成器中的设计人员自动化 (Designer Automation) 功能进行 I/O 连接。

如未指定赛灵思开发板为目标板，您需要为目标器件指定 I/O 连接。请参阅 《Vivado Design Suite UltraFast 设计方法指南》(UG949) [参照 7] 中的链接。

如果在开发进程中使用了您的定制板，建议您创建一个电路板文件，里面详述电路板连接的情况，并在 IP 集成器内实现设计自动化功能，这将极大地简化电路板级连接。有关电路板文件的详情，请通过链接请参阅 《Vivado Design Suite 用户指南：系统级设计输入》(UG895) [参照 9] 中的“使用 Vivado Design Suite 电路板流程”。

项目创建后，使用 Flow Navigator 中的“Create Block Design”按钮，打开 IP 集成器并创建一个新的块设计。在 IP 集成器窗口中，指定您 IP 资源库的源并使用“Add Ip”**按钮开始进行**按钮开始进行 shell 的组合。

shell 完成后，使用 `write_bd_tcl` 命令把整个块设计保存为 Tcl 脚本。脚本包含从头开始重新生成模设计所需的一切。块设计和 Vivado 项目已保存并准备好用于下级验证和系统开发。

在 Documentation Navigator 的“Design Hubs”（设计中心）标签中提供了有关管脚分配、 IP 集成器以及其它功能的进一步信息。如需了解更多信息，请参阅“使用 Documentation Navigator”。

Shell 验证

完成 shell 设计创建后就可以进行 shell 验证。在验证过程中， shell 设计会重新打开，并将验证 IP 添加到设计中，以确认接口正常工作。

shell 验证项目

验证 shell 设计的第一步是使用以下两个选项之一创建新的验证项目。

- 打开用于 shell 设计的 Vivado 项目，使用“File > Save Project As”在新项目中保存 shell 设计。
- 创建一个新的 Vivado RTL 项目 （无 RTL 源），以及相同的目标器件或电路板。然后选择“Create Block Design”，并在控制台中将使用 `write_bd_tcl` 保存的 Tcl 脚本做为源，在新项目中重新生成 shell 块设计。

可能需要使用多个验证项目来确保验证设计的复杂性可控。下图所示的即为 shell 示例验证设计。本示例仅测试单个接口。

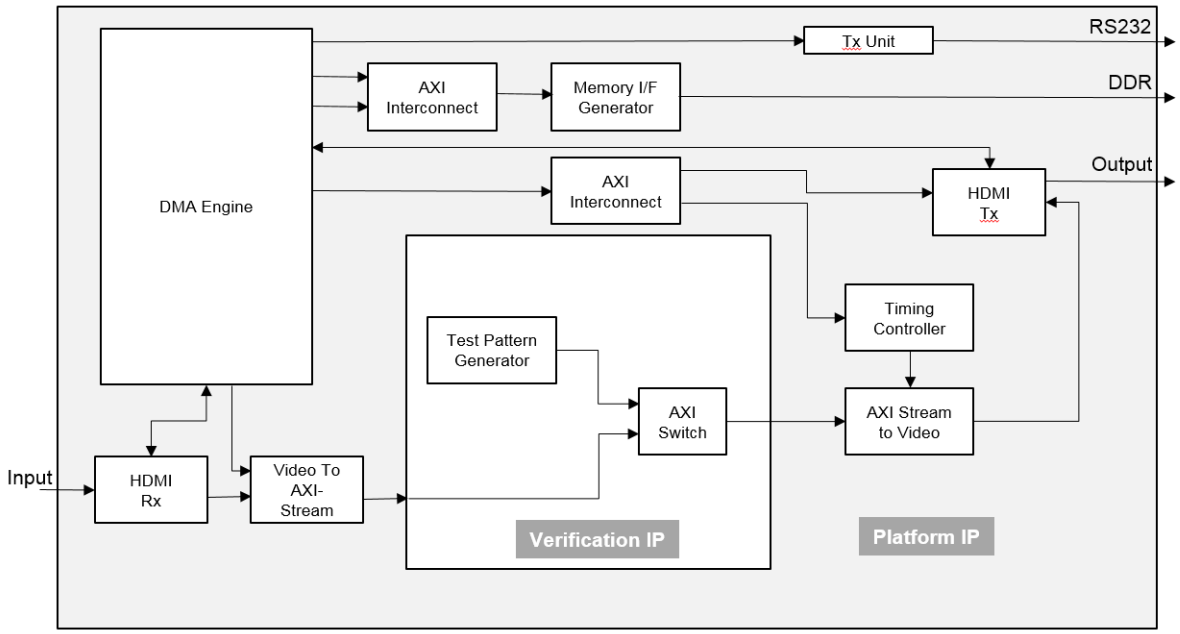

图 3-2：shell 验证示例

验证 IP

将验证 IP 从 Vivado IP 目录中添加到 shell 设计，以验证设计。

本指南中讨论的所有技巧都可用于开发验证 IP：RTL、 System Generator、或基于 C 语言的 IP。下面的示例显示如果采用标准 AXI 接口 IP，如何使用一个小型 C 语言文件快速创建例如 AXI4-Stream 流接口上包含 N 个样本输出的一个 HANN 窗口。如下面代码示例所示，只要将接口指令从 `axis` 改为 `m_axi`，就能够实现一个 AXI 存储器映射接口：

```
void verify_IP_Hann(float outdata[WIN_LEN]) {
// Specify AXI4-Stream output
#pragma HLS INTERFACE axis port=outdata
// Alternative output AXI4M (commented out)
//#pragma HLS INTERFACE m_axi port=outdata

    float coeff[WIN_LEN];
    coeff_loop:for (int i = 0; i < WIN_LEN; i++) {
       coeff[i] = 0.5 * (1.0 - cos(2.0 * M_PI * i / WIN_LEN));
    }

    winfn_loop:for (unsigned i = 0; i < WIN_LEN; i++) {
      outdata[i] = coeff[i];
     }
}
```

如需了解如何使用 Vivado HLS 创建其它 IP 之间的接口块，请参阅 《使用 Vivado IP 集成器集成基于 AXI4 的 IP 应用指南》(XAPP1204) [参照 10]。

验证 shell

如果在仿真源中添加了顶层测试平台，在进行 FPGA 编程前就能够通过仿真验证 shell 设计。

使用 RTL 仿真进行 shell 验证需要创建 RTL 测试平台。使用该测试平台对完全集成的设计进行验证。如果使用多个验证项目来验证 shell，应扩展该测试平台，以验证所有接口。

为验证 FPGA 上的具体接口，可在设计中添加额外的信号级调试探针。

在块设计中工作时，右键单击菜单便可轻松标记网络用于调试。在硬件运行中，可对标记用于调试的信号进行分析：设计中添加了 ILA 核，用以从 FPGA 采集和扫描出信号，以供分析。请参阅 《Vivado Design Suite 用户指南：采用 IP 集成器设计 IP 子系统》(UG994) [参照 8] 中的链接。

最终设计随后通过 Vivado 设计流程处理为比特流。shell 验证完毕后，shell 设计的任何修改都应传回最初的的源 shell 设计项目，但验证 IP 的修改除外。至此， shell 设计已准备就绪，可用于核设计 IP 的集成。

XILINX

第 4 章

基于 C 语言的 IP 开发

简介

在高生产力设计流程中，生成核设计 IP 的主要方式是使用基于 C 语言的 IP 和通过高层次综合 (HLS) 把 C 语言代码转化为 RTL。基于 C 语言的 IP 开发流程提供了下列优势：

- C 语言验证提供的一流仿真速度
- 自动生成时序精确的经优化 RTL
- 能够使用库中的现有 C 语言 IP
- 能使用 IP 集成器轻松地把生成的 RTL IP 集成到完整的系统中

本章讨论关于基于 C 语言的 IP 如何创建、验证、综合、分析、并将其封装为可供 IP 目录使用的 IP。实现这一工作的方法就是 Vivado® 高层次综合 (HLS)，它是一种由 Vivado Design Suite 提供的工具。

Vivado HLS 设计流程见下图所示。设计流程步骤包括：

1. 编译、执行 （仿真）和调试 C 语言算法。

   注释：在高层次综合中，运行编译后的 C 语言程序被称为 C 语言仿真。运行 C 语言程序，仿真该功能，以验证算法功能正常。
2. 综合 C 语言程序为 RTL 设计实现，可以选择性地使用用户优化指令。
3. 生成综合性报告并分析设计。
4. 使用按钮式流程验证 RTL 设计实现。
5. 将 RTL 设计实现封装为一套选定的 IP 格式。

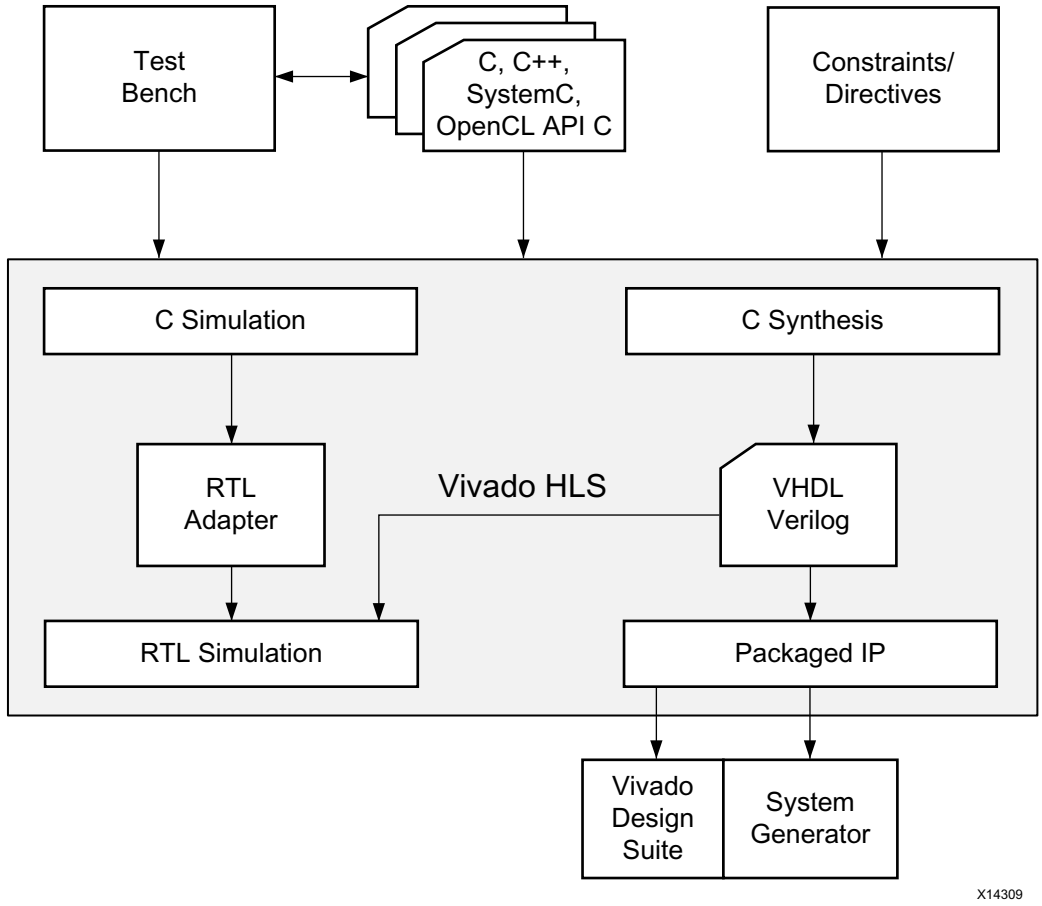

图 4-1：Vivado HLS 设计流程

有关 Vivado HLS 的详情请参阅《Vivado Design Suite 用户指南：高层次综合》(UG902) [参照 2]。本章介绍的是一种高效地使用 Vivado HLS 的方式。

快速 C 验证

与在 RTL 中仿真相比，在 C 语言中仿真相同的算法速度可快上数倍。

以标准的视频算法为例。典型的 C 语言视频算法先处理完整的视频数据帧，然后把输出图像与基准图像做比较，确认结果的正确性。这个算法的 C 语言仿真一般耗时 10 到 20 秒。 RTL 实现的仿真通常需要几个小时到一天 （数天）不等，具体取决于帧的数量和设计的复杂性。

借助软件仿真速度的优势，越多采用 C 语言进行设计开发，您的生产效率就越高。设计人员正是在这一层面上开展实际的设计工作：调整算法、数据类型、比特宽度以验证和确认设计的正确性。

该流程的其余部分属于开发工作：使用工具链在 FPGA 中实现正确的设计。Vivado Design Suite 和高层次生产力设计方法提供的优势在于为设计流程带来了高度自动化。

在初步 FPGA 设计实现后，马上创建一个完整的新比特流为 FPGA 编程的做法并非罕见。使用第 5 章"系统集成"中介绍的脚本化流程，比开展全系统 RTL 仿真耗时更少。

为最大限度提升基于 C 语言的 IP 流程的生产力，应了解以下几点：

C 测试平台

每一个 C 语言程序的顶层都是 `main()` 函数。Vivado HLS 用于综合 `main()` 以下的每一个单项函数。需要 Vivado HLS 综合的函数称之为“设计函数” (Design Function)。详情请参阅图 4-2。

- 设计函数下的所有函数都通过 Vivado HLS 综合。
- 设计函数层级外的全部内容被称为“ C 测试平台”。

C 测试平台包括 `main()` 下的所有 C 语言代码，用于为设计函数提供输入数据以及从设计函数接受输出数据以确认其准确性。

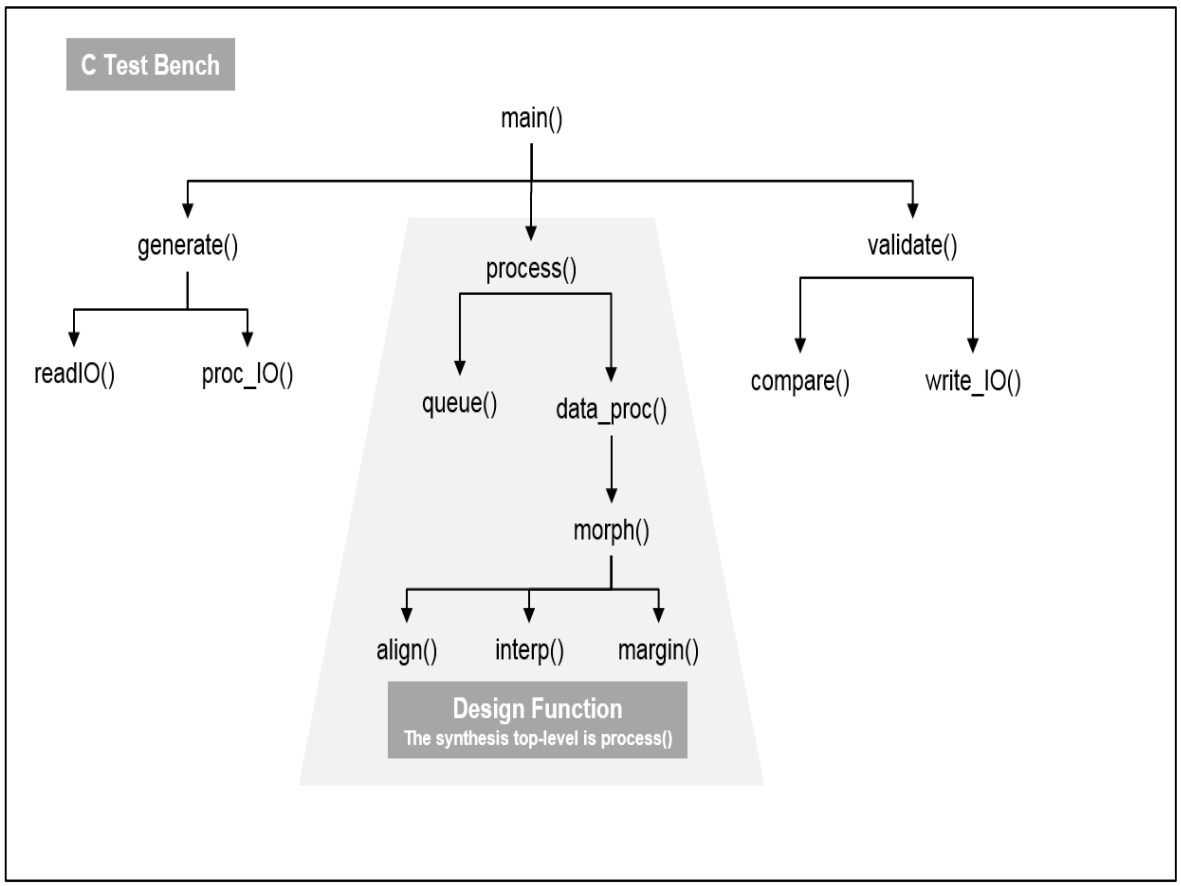

图 4-2：C 测试平台

Vivado HLS 设计流程中新用户可能犯下的最大错误是不使用 C 测试平台和不执行 C 语言仿真就综合自己的 C 语言代码。这一情况突出体现在下面的代码中。在本嵌套循环示例中有哪些错误？

```
#include "Nested_Loops.h"

void Nested_Loops(din_t A[N], dout_t B[N]) {

  int i,j;
  dint_t acc;

  LOOP_I:for(i=0; i < 20; i++){
    LOOP_J: for(j=0; j < 20; j++){
      if(j=0) acc = 0;
        acc += A[i] * j;
      if(j=19) B[i] = acc / 20;
    }
  }
}
```

该代码未能综合成预计的结果，因为条件语句评估为 FALSE 且 J 在 LOOP_J 第一次迭代结束设置为 19。该条件语句的正确表达应为 j==0 且 j==19 （使用 == 取代 =）。前述代码示例能够毫无问题地编译、执行和综合。但是如果只是草率地目测评估该代码，该代码将无法发挥预期的作用。

在开发人员每天不断使用 C/C++、Perl、Tcl、Python、Verilog 和 VHDL 语言中的一种或多种的时代，不仅难以察觉此类小错误，更难以发现功能性错误，而且在综合后发掘它们难度极大、极为耗时。

C 测试平台仅仅是一个调用需要综合的 C 语言函数的程序，用于提供测试数据和测试输出的正确性。它可以在综合前编译和运行并且在综合前验证预期结果。

您在一开始可能会认为直接进行综合能节省时间。但在您的设计方法中使用 C 测试平台带来的好比创建测试平台所用的时间更有价值。

自检测试平台

Vivado HLS 支持在综合前开展 C 语言仿真，以验证 C 语言算法。同时支持在综合后进行 C/RTL 协同仿真，以验证 RTL 设计实现。在两种情况下， Vivado HLS 均使用函数 main() 的 return 值确认结果的正确性。理想的 C 测试平台应有下面的代码示例中所示的结果检查属性。该函数供综合使用的输出保存在文件 results.dat 中并与正确和预期的结果比对，也就是本示例中所谓的“理想”结果。

```
int main () {
  ...
  int retval=0;
  fp=fopen("result.dat","w");
  ...
  // Call the function for synthesis
  loop_perfect(A,B);

  // Save the output results
  for(i=0; i<N;++i) {
    fprintf(fp, "%d \n", B[i]);
  }...

  // Compare the results file with the golden results
  retval = system("diff --brief -w result.dat result.golden.dat");
  if (retval != 0) {
    printf("Test failed  !!!\n");
    retval=1;
  else {
    printf("Test passed !\n");
  }

  // Return 0 ONLY if the results are correct
```

```
    return retval;
}
```

在本 Vivado HLS 设计流程中，函数 `main()` 的 `return` 代表下述意义：

- 零值：结果正确。
- 非零值：结果不正确。

建议：由于系统环境（例如 Linux、 Windows 或 Tcl）用于解读 `main()` 函数的返回值，赛灵思建议出于移植性和安全性考虑，把返回值约束为 8 位范围。

通过使用自检测试平台，您无需创建测试平台来验证 Vivado HLS 的输出的正确性。用于 C 语言仿真的测试平台在 C/RTL 联合仿真过程中也会自动使用，然后用该测试平台验证综合后的结果。

C 语言中有多种途径检查结果的有效性。在上面的示例中，该函数供综合使用的输出保存为文件 `result.dat`，并与含预期结果的文件做比较。这些结果还可以与未标记成用于综合的相同函数做比较（测试平台运行时在软件内执行）或与测试平台计算的值做比较。

重要提示：如果测试平台的函数 `main()` 中没有 `return` 语句， C 语言标准会把 `return` 值设为零。因此 C 语言和 C/RTL 协同仿真往往会报告仿真通过，哪怕结果是错误的。检查结果，只要结果正确就返回零值。

花时间创建自检测试平台可确保 C 语言代码中没有明显错误，无需创建 RTL 测试平台以验证综合得到的输出的正确性。

比特精度数据类型

随 Vivado HLS 提供有任意精度数据类型，可指定任意宽度的变量。例如可以把变量定义为 12 位、22 位或 34 位宽度。使用标准的 C 语言数据类型，这些变量分别应为 16 位、32 位和 64 位。使用标准的 C 语言数据类型往往造成使用不必要的硬件来实现所需的精度。例如仅需要 34 位时却用 64 位硬件来实现。

使用任意精度数据类型的一个更明显的优势是使用这些新的比特宽度仿真 C 语言算法，分析比特精度结果。例如您可能想设计一种 10 位输入和 14 位输出的滤波器，同时您可能判定该设计可使用 24 位累加器。运行 C 语言仿真，在数分钟内即可用数万样本对滤波器进行仿真，确定输出的信噪比是否可接受。您很快就能够判断是否累加器过小，或验证使用更小、更高效的累加器是否仍然提供所需的精度。

重要提示：比特精度 C 语言仿真是验证设计的最快途径。

高生产力方法的特点是使用标准 C 语言数据类型启动您的初始设计，然后确认算法性能是否符合设计。随后移植 C 语言代码，以使用任意精度数据类型。对于这种移植到硬件效率更高的数据类型上的操作，只有在能够用 C 测试平台检查结果的情况下才能够安全且高效地执行。这样您可以迅速地验证较小但更高效的数据类型是否能够胜任。当您已经熟悉任意精度类型的使用，一般就可以在新的 C 语言项目初期就使用任意精度数据类型。

使用 C 测试平台的优势，以及在您的设计方法中不使用它所带来的生产力损失，都非常显而易见。

《Vivado Design Suite 用户指南：高层次综合》(UG902) [参照 2] 中的链接里描述的 Vivado HLS 示例均提供有 C、C++ 或 SystemC 测试平台。这些示例经复制和修改可用于创建 C 测试平台。其中包括使用任意精度数据类型的 C 语言函数。

C 语言对综合的支持

理解综合支持哪些语言是 Vivado HLS UltraFast 设计方法的重要组成部分。 Vivado HLS 为 C、 C++ 和 SystemC 提供全面支持。 C 语言仿真可全面支持，但不能够把每一种描述都综合为等效的 RTL 设计实现。

在审核用于实现在 FPGA 中的代码时应牢记两个原则：

- FPGA 有固定数量的资源。功能必须在编译时固定。硬件中的对象不能动态创建和删除。
- 与 FPGA 的全部通信必须通过输入输出端口进行。 FPGA 不具备底层操作系统 (OS) 或操作系统资源。

不支持的结构

系统调用

综合不支持系统调用。这些调用用于与运行 C 语言程序的操作系统交互。在 FPGA 中不存在需要与之通信的底层操作系统。系统调用的示例有“time()”和“printf()”。

部分常用函数会被 Vivado HLS 自动忽略，因此需要把它们从代码中去除。这些函数是：

- abort()
- atexit()
- exit()
- fprintf()
- printf()
- perror()
- putchar()
- puts()

除了移除任何不受支持的代码之外，还有一种方法是禁止其进入综合。 __SYNTHESIS__ 宏在执行综合时由 Vivado HLS 自动定义。

该宏可用于在运行 C 语言仿真时包含代码，在运行综合时排除该代码。

```
#ifndef __SYNTHESIS__
  // The following code is ignored for synthesis
  FILE *fp1;
  char filename[255];
  sprintf(filename,Out_apb_%03d.dat,apb);
  fp1=fopen(filename,w);
  fprintf(fp1, %d \n, apb);
  fclose(fp1);
#endif
```

注释：只在需要综合的代码中使用 __SYNTHESIS__ 宏。请勿在测试平台中使用该宏，因为 C 语言仿真或 C/RTL 联合仿真不会遵从。

如果需要操作系统提供信息，该数据必须作为实参传递给顶层函数供综合使用。随后就是系统其余部分的任务，将该信息提供给综合后的 IP 块。这一工作一般的完成方法是把该数据端口实现为连接到 CPU 的 AXI4-Lite 接口。

动态对象

动态对象不能综合。函数调用 `malloc()` 和 `alloc()`、预处理器 `free()` 和 C++ `new` 以及 `delete` 能够动态地创建或删除操作系统存储器映射中的存储器资源。FPGA 中可用的唯一存储器资源是块 RAM 和寄存器。块 RAM 是阵列综合时创建的，必须在一个或者数个时钟周期间保持阵列中的值。当在一个或者数个时钟周期内必须保持变量存储的值时，就要创建寄存器。必须使用固定大小的阵列或变量替代任何动态存储器分配。

和用于动态存储器使用的限制一样， Vivado HLS 不支持动态创建或删除的 C++ 对象（用于综合）。这包括动态多态性和动态虚拟函数调用。可能形成新硬件的新函数不能在运行中动态地创建。

出于类似原因，综合中也不支持递归。所有对象在编译时必须有已知的大小。在使用模板时可有限地支持递归。

除了 `std::complex` 等标准数据类型，综合中也不支持 C++ 标准模板库。这些库内包含的函数会广泛的使用动态存储器分配和递归。

SystemC 结构

`SC_MODULE` 不能内嵌套或是从另一个 `SC_MODULE` 衍生。

不支持 `SC_THREAD` 结构 （但支持 `SC_CTHREAD`）。

有限支持的结构

顶层函数

支持使用模板进行综合，但不支持用于顶层函数。

C++ 类对象不能在综合中直接用于顶层。该类必须例化为顶层函数。

综合中支持指针到指针的指向，但如果用作顶层函数的实参就不支持。

指针支持

Vivado HLS 支持原生 C 语言类型间的指针转型，但不支持一般性的指针转型，比如为区别结构类型而在指针间进行转型。

Vivado HLS 支持指针阵列，只要每一个指针指向 scalar 或 scalar 阵列。指针阵列不能指向其他指针。

递归

在 FPGA 中只支持使用模板的递归。综合中执行递归的关键是使用尺寸为 1 的终端类在递归中实现最终调用。

存储器函数

`memcpy()` 和 `memset()` 均受支持，但其限制条件是必须使用 `const` 值。

- `memcpy()`：用于总线突发运行或使用恒值的阵列初始化。 memcpy 函数只能用于把值复制到用于顶层函数的参数或从中将值复制出来。
- `memset()`：用于使用常量设置值进行集初始化。

任何不支持用于综合的代码或任何有限支持用于综合的代码必须在能够综合前加以修改。

关于语言支持的详细介绍请参阅 《Vivado Design Suite 用户指南：高层次综合》(UG902) [参照 2] 中的链接。

使用经硬件优化的 C 语言库

Vivado HLS 为常用的 C 语言函数提供了一定数量的 C 语言库。 C 语言库中提供的函数一般经过预先优化，可确保在综合时得到高性能和高效率设计实现。

《Vivado Design Suite 用户指南：高层次综合》(UG902) [参照 2] 中的链接介绍的“高层次综合 C 语言库”对随 Vivado HLS 提供的所有 C 语言库进行了充分介绍，但作为您方法的一部分，强烈推荐您对 C 语言库中提供的 C 语言函数进行深入了解。

Vivado HLS 包含下列 C 语言库：

- 任意精度数据类型
- HLS 流库
- 数学函数
- 线性代数函数
- 数字信号处理 (DSP) 函数
- 视频函数
- IP 库

理解 Vivado HLS

在点评本指南中后续关于优化基于 C 语言的 IP 的章节之前，必须首先掌握一些关键的 HLS 概念。本节的目的就是对这些概念做简要的总体介绍。

衡量性能

Vivado HLS 根据自身默认的综合行为和约束可迅速创建最理想的设计实现。时钟周期是主要的约束。 Vivado HLS 使用时钟约束，结合目标器件的规格判断每个时钟周期内可完成的操作次数。

在满足时钟频率约束后， Vivado HLS 所使用的性能指标按优化重要性排序是：

- 初始间隔 (II)：这是新输入之间的时钟周期数量。它代表了吞吐能力，以及设计读取下一输入并加以处理的速度。
- 时延：这是指生成输出所需的时钟周期数量。在实现最小间隔后，或是没有明确内部目标的情况下， Vivado HLS 会尽量最小化时延。
- 面积：在实现最小时延后， Vivado HLS 会尽量最小化面积。

在报告性能指标时，报告的是整个函数 （功能）的指标。例如，如果函数使用 scalar 输入， II=3 的意思是该函数每 3 个时钟周期处理 1 个样本。但如果该函数输入的是由 N 元组成的阵列，II=N 的意思是每 N 个时钟周期处理 N 个元素：即每时钟周期处理一个样本的速率。

使用优化指令可指引 Vivado HLS 创建上述指标优先的设计。例如，强制降低一定吞吐量的面积占用或时延。在没有任何优化指令的情况下， Vivado HLS 根据这些目标并使用下面介绍的默认综合行为创建初始设计。

接口综合

顶层函数的实参用可选 I/O 协议综合为数据端口。 I/O 协议指与数据端口相关的一个或多个信号，用于自动同步该数据端口与系统中其他硬件块之间的数据通信。

例如，在握手协议中，数据端口会伴随一个有效端口用于说明何时数据对读取或写入有效以及一个确认端口用于说明数据已经成功读取或写入。

关于 I/O 协议的完整介绍，请参阅 《VivadoDesign Suite 用户指南：高层次综合》(UG902) [参照 2] 中的链接，不过这些接口包含 AXI、 AXI4-Stream 和 AXI4-Lite 接口，可使用 IP 集成器方便地把 IP 集成到系统中。

此外， I/O 协议默认条件仅为顶层函数自身而实现。该协议控制 IP 开始运行的时间，明确 IP 完成运行的时间或已经准备好接受新的输入数据。该可选 I/O 协议可实现为 AXI4-Lite 接口以便用微处理器控制该设计。

函数综合

函数被综合为最终 RTL 设计中的层级块。 C 语言代码中的每一个函数在最终 RTL 中都用一个唯一的块代表。一般情况下优化终止于函数边界。部分优化指令具有递归选项，或者其行为能让该指令跨越函数边界生效。

使用优化指令可以内联函数。这样可以永久性地去除函数层级，实现更加优异的逻辑优化。函数还可以流水线化，提高它们的吞吐量性能。

函数在日程中安排为尽早执行。下面的示例是两个函数 `foo_1` 和 `foo_2`。

```
void foo_1 (a,b,c,d,*x,*y) {
  ...
  func_A(a,b,&x);
  func_B(c,d,&y);
}
```

在函数 `foo_1` 中，函数 `func_A` 和 `func_B` 之间不存在数据相依性。虽然它们在 C 语言代码中是顺序出现， Vivado HLS 实现的架构让这两个函数在第一个时钟周期同时开始处理数据。

```
void foo_2 (a,b,c,*x,*y) {
  int *inter1;
   ...
   func_A(a,b,&inter1,&x);
   func_B(c,d,&inter1,&y)
}
```

在函数 `foo_2` 中，函数之间有数据相依性。内部变量 `inter1` 由 `func_A` 传递给 `func_B`。在本示例中 Vivado HLS 必须调度函数 `func_B` 在 `func_A` 执行完成后才开始执行。

循环综合

循环在默认下保持“收起”状态。这意味着 Vivado HLS 只综合循环体中的逻辑一次，然后顺序执行该逻辑，直至抵达循环终端条件。循环可以“展开”以便所有操作并行进行。但这样会建立循环硬件的多个复本。它们经流水线化可以提升性能。

循环一般调度为按顺序执行。在下面的示例中循环 SUM_X 和 SUM_Y 之间没有相依性，但它们一直按在代码中出现的次序进行调度。

```
#include "loop_sequential.h"

void loop_sequential(din_t A[N], din_t B[N], dout_t X[N], dout_t Y[N],
              dsel_t xlimit, dsel_t ylimit) {

     dout_t X_accum=0;
     dout_t Y_accum=0;
     int i,j;

     SUM_X:for (i=0;i<xlimit; i++) {
          X_accum += A[i];
          X[i] = X_accum;
}

     SUM_Y:for (i=0;i<ylimit; i++) {
          Y_accum += B[i];
          Y[i] = Y_accum;
     }
}
```

图 4-2：时序循环

逻辑综合

默认条件下函数和循环中的逻辑一直综合为尽早执行。 Vivado HLS 总会尽量最小化时延和实现设计。 C 语言代码中的运算符，如 +、 * 和 /，都综合为硬件核。 Vivado HLS 自动选择最合适的核以实现综合目标。可使用优化指令 RESOURCE 明确地指定使用哪个硬件核来实现特定操作。

阵列综合

默认条件下 Vivado HLS 把阵列综合为块 RAM。

在 FPGA 中，块 RAM 提供在由 18K 位原语元组成的块中。每个块 RAM 根据需求使用特定数量的 18K 原语元实现该阵列。例如由 1024 种 int 类型组成的阵列需要 1024 * 32 位 = 32768 位 块 RAM。相当于需要 32768/18000 = 1.8 18K 块 RAM 原语来实现该块 RAM。虽然 Vivado HLS 报告每个阵列综合为一个块 RAM，该块 RAM 可能包含多个 18K 原语块 RAM 元。

在默认条件下 Vivado HLS 不会试图把较小的块 RAM 集合成一个大型块 RAM 或是把大型块 RAM 分区成较小的块 RAM。但是在使用优化指令时可能出现这种情况。 Vivado HLS 可能自动把小阵列分区为单独的寄存器，以提升结果质量。

Vivado HLS 根据综合目标自动判断是否使用单端口或双端口块 RAM。例如，如果有助于最小化时间间隔或时延， Vivado HLS 会使用双端口块 RAM。为明确地指定是否使用单端口或双端口块 RAM，用户可以使用 RESOURCE 优化指令。

流水线函数、循环和任务

实现高性能设计的关键在于使用 PIPELINE 和 DATAFLOW 优化指令来流水线化函数、循环和任务。

下图是说明流水线化的概念解释。在不使用流水线化的情况下，操作会顺序执行直至函数完成。然后开始函数的下一次执行或下一个传输事务。通过流水线，一旦硬件资源可用，下一个传输事务就会即刻启动。

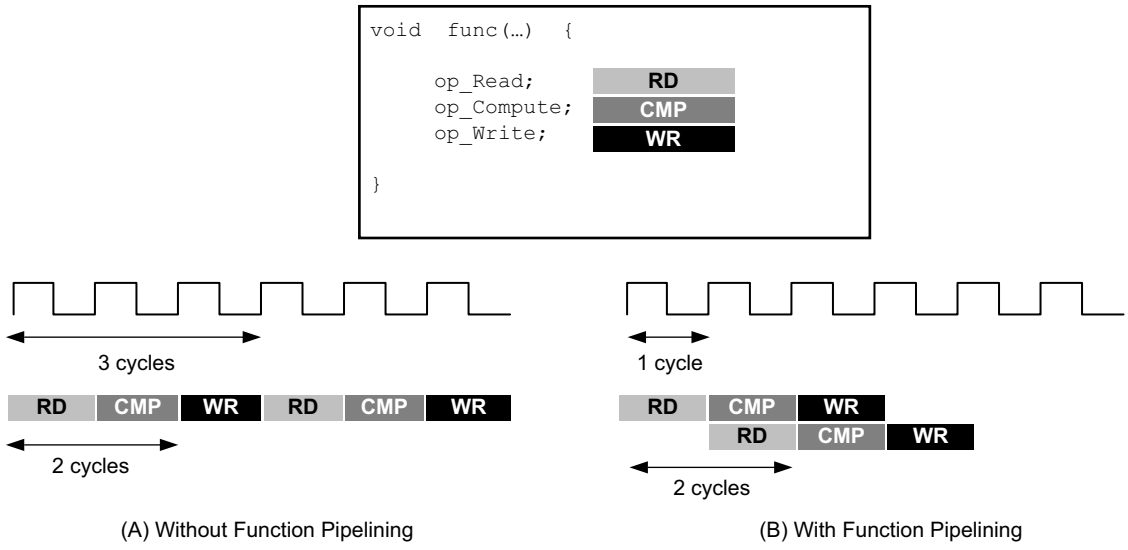

图 4-3：流水线行为

PIPELINE 指令可在函数或循环上使用，以最小的面积开销提升吞吐能力 （最小化 II）。

函数和循环均视为任务。 DATAFLOW 指令用于“流水线”任务，让它们在数据关联关系允许的情况下同时执行。

图 4-4 所示即为任务流水线化的概念视图。综合完成后，默认行为是先执行和完成 func_A，然后是 func_B，最后是 func_C。不过您可以使用 DATAFLOW 优化指令调度每个函数，只要数据可用就即刻执行。

在本示例中原始函数的时延和时间间隔为 8 个时钟周期。在使用数据流 (dataflow) 优化后，时间间隔缩短到仅 3 个时钟周期。本例中所示的任务是函数，但用户也可以在函数之间、函数与循环之间以及循环之间执行数据流优化。

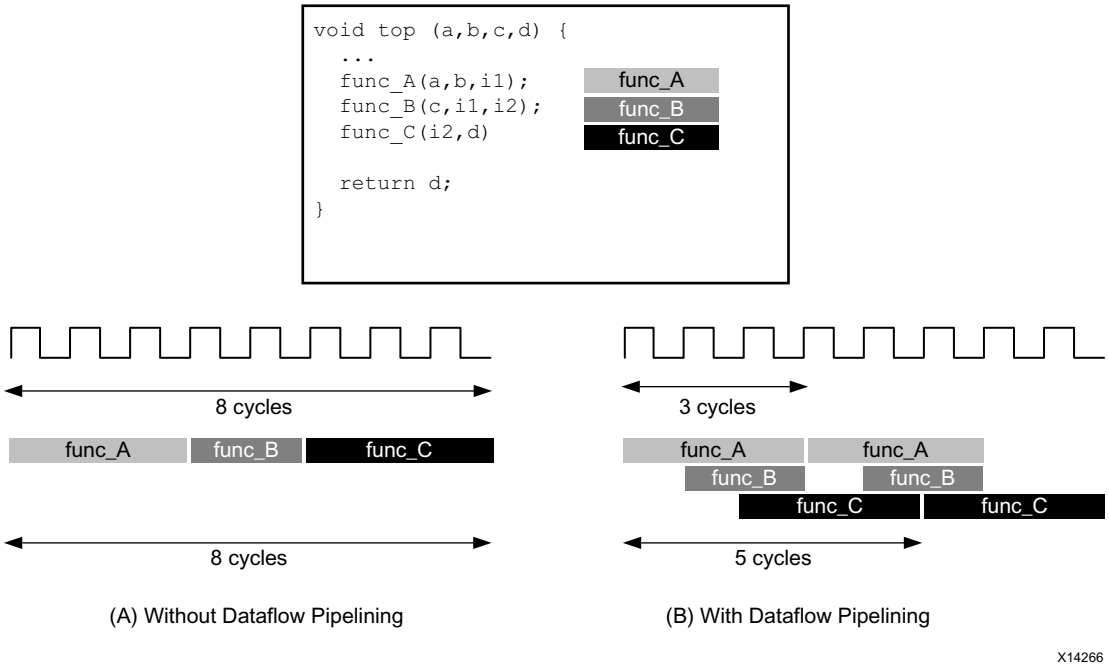

图 4-4：数据流优化

Vivado HLS 资源

Documentation Navigator 中的 Vivado HLS 设计中心为进一步了解 Vivado HLS 提供了方便的渠道：

- QuickTake 视频讲解了具体操作
- 关于该设计流程各个方面的教程
- Vivado HLS 用户指南
- 多个应用指南

如需了解有关的设计中心更多信息，请参阅“使用 Documentation Navigator”。

优化方法

除前述章节讨论的默认综合行为， Vivado HLS 还提供了一定数量用于引导综合得出所需结果的优化指令和配置。本节将介绍用于优化设计以实现高性能的一般性方法。

在使用 Vivado HLS 优化设计时可能面临多重目标。该方法假定您需要创建的是性能最高的设计，每个时钟周期处理一个新输入数据样本，在考虑用于降低时延和节省资源的优化之前，先开展速度优化。

下一节“HLS 优化方法”将探讨如何把本节介绍的方法应用于不同的 C 语言代码架构。

关于本节介绍的各种优化的详细解释，请参阅 《Vivado Design Suite 用户指南：高层次综合》(UG902) [参照 2]。

- 链接“管理接口”。
- 链接“设计优化”。

强烈推荐在审核具体优化的详细内容之前先了解本方法并建立对高层次综合优化的全局观念。

HLS 优化方法

Vivado HLS 的优化方法请见图 4-5。首先验证 C 语言代码是否功能正常，这尤其重要。其余的步骤将在下文中具体介绍，可大致分为：确定接口、流水线化设计、通过优化数据结构解决阻碍理想的流水线化的问题、然后解决任何时延和面积问题。

图 4-5：HLS 优化方法

下面是优化指令的完整清单。清单左侧显示的是 Tcl 命令，右侧是可以直接包含在 C 语言代码中的等效编译指示 (pragma)。

```
set_directive_allocation           - Directive ALLOCATION
set_directive_array_map            - Directive ARRAY_MAP
set_directive_array_partition      - Directive ARRAY_PARTITION
set_directive_array_reshape        - Directive ARRAY_RESHAPE
set_directive_data_pack            - Directive DATA_PACK
set_directive_dataflow             - Directive DATAFLOW
set_directive_dependence           - Directive DEPENDENCE
set_directive_expression_balance   - Directive EXPRESSION_BALANCE
set_directive_function_instantiate - Directive FUNCTION_INSTANTIATE
set_directive_inline               - Directive INLINE
set_directive_interface            - Directive INTERFACE
set_directive_latency              - Directive LATENCY
set_directive_loop_flatten         - Directive LOOP_FLATTEN
set_directive_loop_merge           - Directive LOOP_MERGE
set_directive_loop_tripcount       - Directive LOOP_TRIPCOUNT
set_directive_occurrence           - Directive OCCURRENCE
set_directive_pipeline             - Directive PIPELINE
set_directive_protocol             - Directive PROTOCOL
set_directive_reset                - Directive RESET
set_directive_resource             - Directive RESOURCE
set_directive_stream               - Directive STREAM
set_directive_top                  - Directive TOP
set_directive_unroll               - Directive UNROLL
```

通过配置可修改默认的综合行为。配置没有等效的编译指示。在 GUI 中使用菜单“Solution > Solution Settings > General”来设置配置。可用配置的完整清单如下：

```
config_array_partition            - Config the array partition
config_bind                       - Config the options for binding
config_compile                    - Config the optimization
config_dataflow                   - Config the dataflow pipeline
config_interface                  - Config command for io mode
config_rtl                        - Config the options for RTL generation
config_schedule                   - Config scheduler options
```

拥有所有的优化指令和综合配置固然好。掌握使用它们的方法却更好。

步骤 1：初始优化

下表显示的是您应首先考虑将其加入设计内的指令。

表 4-1：优化策略步骤 1：初始优化

指令和配置	描述
INTERFACE	指定如何根据功能描述创建 RTL 端口。
DATA_PACK	把结构 (struct) 的数据字段打包到有更大字宽度的单一 scalar 中。
LOOP_TRIPCOUNT	用于带有变量的循环。为循环迭代计数提供估算。这对综合没有影响，只对报告功能有影响。
Config Interface	该配置控制着与顶层函数实参不存在关联的 I/O 端口，能够从最终 RTL 中去除未使用的端口。

该设计接口一般用系统中的其他块定义。因为 I/O 协议的类型有助于确定综合的结果，建议在进行设计优化之前使用 INTERFACE 指令加以明确。

如果该算法以流方式访问数据，可以考虑使用某个流协议来确保高性能运行。

提示：如果 I/O 协议完全被外部块固定，永远无法修改，可考虑将 INTERFACE 指令作为编译指示直接插入到 C 语言代码中。

当在顶层实参清单中使用结构时，它们会被分解为单独的元，且结构的每个元都实现为单独的端口。在某些情况下可使用 DATA_PACK 优化把整个结构实现为单个数据字，从而得到单个 RTL 端口。如果结构包含大型阵列应谨慎处理。阵列的每个元都实现在数据字中，可能导致非常宽的数据端口。

设计在综合后一个常见问题是报告文件把时延和时间间隔显示为问号“? ”而非数值。如果设计含有带变量循环绑定的循环， Vivado HLS 就不能确定时延且用“?”来说明这种情况。

要解决此问题，请使用分析视角或综合报告定位综合无法报告数值的最底层循环，然后使用 LOOP_TRIPCOUNT 指令施加一个估算的跳脱计数 (tripcount)。这样就可以报告时延和时间间隔值，也可以比较不同优化方法下的解决方案。

注释：带变量绑定的循环不能完全展开，且阻碍层级中位于其上方的函数和循环的流水线化。这个问题将在下一节中讨论。

最后，全局变量一般是在供综合的函数范围内读取或写入，无需用作最终 RTL 设计中的 I/O 端口。如果全局变量用于向 C 语言函数传递数据或是从 C 语言函数传出数据，可以考虑使用接口配置功能将其配置为 I/O 端口。

步骤 2：性能流水线

创建高性能设计的下一阶段是流水线化函数、循环和任务。下表显示的是用户可用于流水线化的指令。

表 4-2：优化策略步骤 2：性能流水线

指令和配置	描述
PIPELINE	通过让循环中或函数中的操作同时执行，降低初始化时间间隔。
DATAFLOW	实现任务层面的流水线化，让函数和循环同时执行。用于最小化时间间隔。
RESOURCE	指定用于在 RTL 中实现变量（阵列、算术运算或函数实参）的资源（核）。
Config Compile	让循环根据自身的迭代计数自动流水线化。

在优化流程的这个阶段，用户应尽量多地创建同步操作。您能够对函数和循环应用 PIPELINE 指令。您可在包含函数和循环的层次使用 DATAFLOW 指令，让它们并行工作。

推荐采用自下而上的操作方法并注意下列情况：

- 部分函数和循环内含子函数。如果子函数没有流水线化，位于其上的函数在流水线化后可能性能提升有限。未流水线化的子函数将成为制约因素。
- 部分函数和循环内含子循环。当用户使用 PIPELINE 指令时，该指令会自动在其下方的层级中展开所有的循环。这样会产生大量的逻辑。用下方的层级将循环流水线化，这或许更为合理。
- 带变量帮定的循环不能展开。层级中任何位于此类循环之上的循环和函数无法流水线化。为解决这一问题，可先流水线化这些循环，然后使用 DATAFLOW 优化来最大限度地提升包含此类循环的函数的性能。此外也可重新写入循环，去除变量绑定。

优化流程中的基本策略是尽可能地将各项任务 （函数与循环）实现流水线化。有关有哪些函数和哪些循环到流水线，以及在哪里应用 DATAFLOW 指令的详细信息，请参考“优化策略”。

对具有大量循环或大量嵌套循环的设计，编译配置功能提供了一种根据设计的循环迭代计数对设计中的全部循环自动流水线化的途径。如需了解更多信息，请参阅 《Vivado Design Suite 用户指南：高层次综合》(UG902) [参照 2] 中的链接。

虽然不常用到，您也可以在运算符层面应用流水线。例如 FPGA 中的线路布线会引发大量的和预计之外的延迟，让设计难以实现在要求的时钟频率上。在这种情况下，您可使用 RESOURCE 指令流水线化专门的操作，比如乘法器、加法器和块 RAM。

RESOURCE 指令可指定使用哪些硬件核在 C 语言代码中实现这些运算。如果指定使用大于 1 的时延实现某种资源，这会导致 Vivado HLS 为该运算使用额外的流水线级。 RTL 综合能利用这些额外的流水线级提升总体时序。

下列运算支持流水线化的设计：

- 有多级 (*nS) 核可用的标准算术运算
- 浮点运算
- 用块 RAM 实现的阵列

步骤 3：优化性能结构

C 语言代码可包含防止函数或循环根据性能要求而被流水线化的描述。在某些情况这需要进行代码修改，但在大多数情况下这些问题可以使用其他优化指令解决。

下面的示例即是使用优化指令提升流水线性能的情况。在这个前期示例中，循环中添加了 PIPELINE 指令，以提升循环的性能。

```
#include "bottleneck.h"

dout_t bottleneck(din_t mem[N]) {

  dout_t sum=0;
  int i;

  SUM_LOOP: for(i=3;i<N;i=i+4)
#pragma HLS PIPELINE
  sum += mem[i] + mem[i-1] + mem[i-2] + mem[i-3];

  return sum;
}
```

当上述代码被综合时，会输出下面的消息：

```
INFO: [SCHED 61] Pipelining loop 'SUM_LOOP'.
WARNING: [SCHED 69] Unable to schedule 'load' operation ('mem_load_2',
bottleneck.c:62) on array 'mem' due to limited memory ports.
INFO: [SCHED 61] Pipelining result: Target II: 1, Final II: 2, Depth: 3.
I
```

如果流水线化不能满足所需性能，解决问题的关键在于采用分析角度核查设计。关于使用分析视角的详情请参阅《Vivado Design Suite 用户指南：高层次综合》(UG902) [参照 2] 中的链接。分析视角中该设计示例的视图如下所示。

- 存储器 （块 RAM）访问在该图中以高亮显示。这些对应的是上述代码中的阵列 mem。
- 每次访问占用两个周期：一个用于生成地址，一个用于读取数据。
- 由于一个块 RAM 最多只有两个数据端口，周期 C1 中只能启动两次存储器读取。
- 第三次和第四次存储器读取只能在周期 C2 中开始。
- 下一个存储器读取集合中排在最前读取的可在周期 C3 中开始。这意味着该循环只能有 II=2：只能每两个周期读取一次循环的下一个输入集合。

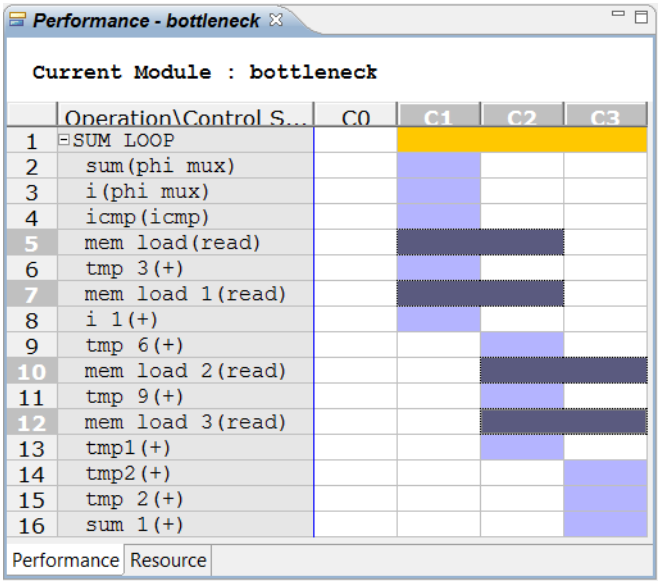

图 4-6：端口数过少导致流水线失败

可以通过对 mem 阵列使用 ARRAY_PARTITION 指令来解决存储器端口限制问题。该指令用于把阵列分区为较小阵列，提供更多数据端口，并提升数据结构，实现性能更高的流水线。

使用下面所示的附加指令，阵列 mem 分区为两个存储器，这样所有四次读取都能够在一个时钟周期内完成。在分区阵列时可以有多种选择。在这里因数为 2 的循环分区可确保第一个分区包含来自原始阵列的元 0、 2、 4，第二个分区包含元 1、 3、 5 等。使用双端口块 RAM，就可以在一个时钟周期中读取元 0、 1、 2 和 3。

```
#include "bottleneck.h"

dout_t bottleneck(din_t mem[N]) {
#pragma HLS ARRAY_PARTITION variable=mem cyclic factor=2 dim=1

  dout_t sum=0;
  int i;

  SUM_LOOP: for(i=3;i<N;i=i+4)
#pragma HLS PIPELINE
  sum += mem[i] + mem[i-1] + mem[i-2] + mem[i-3];

  return sum;
}
```

在尝试流水线化循环和函数时可能会遇到其他的此类问题。下表列出的指令通过帮助减少数据结构中的瓶颈，从而有望解决这些问题。

表 4-3：优化策略步骤 3：优化性能结构

指令和配置	描述
ARRAY_PARTITION	把大型阵列分区为多个较小阵列或分区为单独的寄存器，以提升对数据的访问，消除块 RAM 瓶颈。
DEPENDENCE	用于提供能克服循环承载关联性，让循环流水线化（或使用较低时间间隔流水线化）的额外信息。
INLINE	内联函数，消除所有函数层级。用于跨越函数边界实现逻辑优化，通过减少函数调用开销提升时延/时间间隔。

表 4-3：优化策略步骤 3：优化性能结构 （续）

指令和配置	描述
UNROLL	展开循环，创建多个独立操作而非单个操作集。
Config Array Partition	该配置用于判断包括全局阵列在内的阵列分区方式，以及分区是否会影响阵列端口。
Config Compile	控制综合专用优化功能，例如自动循环流水线化和浮点数学优化。
Config Schedule	确定在综合调度阶段使用的工作量、输出消息的长度，并指明为实现定时是否应在流水线任务中放宽 II。
CONFIG_UNROLL	允许自动展开低于规定循环迭代次数的所有循环。

除 ARRAY_PARTITION 指令之外，用于阵列分区的配置也可用于自动划分阵列。

用于编译的配置可用于自动流水线化循环层级。在流水线化循环时，可能需要用 DEPENDENCE 指令去除暗含关联性。此类关联性由消息 SCHED-68 报告。

```
@W [SCHED-68] Target II not met due to carried dependence(s)
```

INLINE 指令用于移除函数边界。这样可以把逻辑或循环上移一个层级。把逻辑包含在其上方的函数中。通过提高循环的层级，能够更加方便地把一系列循环与其他循环一同“数据流化”，这或许是对函数内逻辑实现流水线化的更高效的方法。

在循环不能使用要求的初始化时间间隔流水线化的时候，可考虑使用 UNROLL 指令。如果某个循环只能流水线化为 II=4，它将把系统中的其他循环和函数制约在 II=4 的水平。在某些情况下展开循环是值得的，以更多逻辑为代价换取解决潜在瓶颈的优势。

调度配置功能的目的是用于增加调度信息的长度并控制调度中的工作量。在使用“详细” (verbose) 选项时， Vivado HLS 会在调度功能无法满足约束要求时列出关键路径。

一般来说，很少 f 有增强调度工作量即能提升调度的情况，但该选项可用。如果优化指令和配置无法用于提升初始化间隔时间，那么可能需要修改代码。这方面的示例请参阅 《Vivado Design Suite 用户指南：高层次综合》(UG902) [参照 2] 中的链接。

步骤 4：减少时延

当 Vivado HLS 完成初始化时间间隔的最小化后，会自动尽量最小化时延。下表中列出的优化指令能帮助降低或指定特定时延。

在循环和函数流水线化的时候一般不需要使用这些指令，因为在大多数应用中时延并非关键，吞吐能力通常才是关键。如果循环和函数没有流水线化，吞吐能力就会受时延所限，因为在任务完成之前不会开始读取下一个输入集合。

表 4-4：优化策略步骤 4

指令	描述
LATENCY	用于指定最小和最大时延约束。
LOOP_FLATTEN	把嵌套循环分解为具备改进时延的单循环。
LOOP_MERGE	融合连续循环，以降低总体时延，增加共享和提升逻辑优化。

LATENCY 指令用于指定要求的时延。循环优化指令用于扁平化循环层级或把多个串行循环融合在一起。由于一般进入和离开循环要消耗一个时钟周期，这样做可以提升时延。循环间过渡数量越少，设计完成运行消耗的时钟周期数量就越少。

步骤 5：缩小面积

在满足要求的性能目标 （或 II）后，下一步是在保持性能不变的情况下缩小面积。

如果使用 DATAFLOW 优化后 Vivado HLS 无法判定设计中的任务是否在流传输数据， Vivado HLS 会使用交替缓存 (ping-pong buffer) 实现数据流任务之间的存储器通道。如果设计已经流水线化且数据正在从一个任务流向下一个任务，使用数据流配置功能 `config_dataflow` 把默认存储器通道中使用的交替缓存转换为 FIFO 缓存，就可以显著地缩小面积。随后可把 FIFO 深度设置为需要的最小值。

数据流配置功能 `config_dataflow` 指定了所有存储器通道的默认实现方式。您可以使用 STREAM 指令指定把哪些单独的阵列实现为块 RAM，把哪些实现为 FIFO。

如果设计使用 `hls::stream` I/O 协议实现， FIFO 的存储器通道默认深度为 1，就无需使用数据流配置。但在任务输出的数据量大于其输入量的情况下，如插值运算，可以使用 STREAM 指令增大 FIFO 的深度。

下表列出了其他指令，可用于最大限度地降低设计实现耗费的资源。

表 4-5：优化策略步骤 5

指令	描述
ALLOCATION	为使用的操作、核和函数数量设定限额。这样会强制共享硬件资源并可能增大时延。
ARRAY_MAP	把多个较小的阵列结合成单个大型阵列以帮助减少块 RAM 资源数量。
ARRAY_RESHAPE	把阵列从多元型阵列重塑为字宽度更大的阵列。用于在避免使用更多块 RAM 的情况下提升块 RAM 访问。
LOOP_MERGE	融合连续循环，以降低总体时延，增加共享和提升逻辑优化。
OCCURRENCE	在流水线化函数或循环时使用，用于让位于某位置的代码的执行速度低于其外层函数或循环中的代码执行速度。
RESOURCE	指定将特定的库资源 （核）用于实现 RTL 中的变量 （阵列、算术运算或函数实参）。
STREAM	指定把特定的存储器通道在数据流优化过程中实现为 FIFO 或 RAM。
Config Bind	判断综合绑定阶段的工作量，可用于在全局层面最小化使用的操作数量。
Config Dataflow	该配置用于指定数据流优化中的默认存储器通道和 FIFO 深度。

ALLOCATION 和 RESOURCE 指令用于限制操作数量，选择哪些核 （或资源）供实现操作使用。例如用户可以限制函数或循环只使用一个乘法器并指定使用流水线化乘法器来实现它。绑定配置用于在全局层面限制特定操作的使用。

重要提示：优化指令只适用于它们规定适用的范围内。

如果 ARRAY_PARITION 指令用于提升初始化时间间隔，可以考虑使用 ARRAY_RESHAPE 指令予以取代。ARRAY_RESHAPE 优化执行的是与阵列分区类似的任务，但重塑优化功能把阵列分区创建的元重新结合成有更宽数据端口的单个块 RAM。

如果 C 语言代码内含索引相似的一连串循环，使用 LOOP_MERGE 指令融合这些循环可以实现某些优化。

最后如果流水线区域内的一段代码需要使用低于区域其他部分的初始化时间间隔，可使用 OCCURENCE 指令指示该逻辑能够通过优化采用更低的速度运行。

优化策略

该优化方法普遍适用于所有类型的 C 语言代码。获得高性能设计的关键优化指令是 PIPELINE 指令和 DATAFLOW 指令。本节详细讨论如何将这些指令用于各类 C 语言代码架构。

分区从根本上来说，存在两种类型的 C 语言函数。一类是基于帧的 C 语言函数，一类是基于样本的 C 语言函数。

无论使用哪种方式，两种情况下生成的 RTL IP 都几乎相同，区别在于应用优化指令的方法。选择哪种方式取决于您自身：使用最便于捕获您的描述的方式。

基于帧的 C 语言代码

基于帧的 C 语言代码的概要示例见下。这种编码方式的主要特点是该函数在每个传输事务中处理多个数据样本 （一帧数据）。这里一个传输事务指该 C 语言函数一次完整地执行。

```
void foo(
  data_t in1[HEIGHT][WIDTH],
  data_t in2[HEIGHT][WIDTH],
  data_t out[HEIGHT][WIDTH] {


Loop1: for(int i = 0; i < HEIGHT; i++) {
  Loop2: for(int j = 0; j < WIDTH; j++) {
    out[i][j] = in1[i][j] * in2[i][j];
    Loop3: for(int k = 0; k < NUM_BITS; k++) {
  }
}
}
```

该数据一般以阵列形式提供，但也可提供为指针或 `hls::stream`。使用指针算法可多次访问指针。`hls::stream` 也可在函数内多次访问。

另一个基于帧的编码方式的特点是数据一般使用循环访问和处理。上述代码是这种情况的经典示例。

在试图流水线化任何 C 语言代码时，您应在处理数据样本的层级上布局流水线指令。根据此原则讨论上面示例中的每一个层级，有助于掌握布局流水线指令的最佳做法。

函数级：该函数接受数据帧作为输入 （in1 和 in2）。如果该函数使用 II=1 流水线化，即每个时钟周期读取一个新输入集合，就会告知工具在单个时钟周期内读取 in1 和 in2 的所有 HEIGHT*WIDTH 值。

这很可能不是您所希望的设计。

在应用 PIPELINE 指令后，这一层以下的所有层级中的循环，即本例中 foo 以下的全部内容都必须展开。这就是流水线化的要求：流水线内部不能出现顺序逻辑。这样会创建逻辑的 HEIGHT*WIDTH*NUM_ELEMENT 复本，形成一个大型设计。

这个阵列可以实现为多种类型的接口：

- 块 RAM 接口 （默认）
- AXI 接口
- AXI4-Lite 接口
- AXI4-Stream 接口
- FIFO 接口

由于数据以顺序循环的方式被访问，作为 AXI4-Stream 接口。双向握手或 FIFO 接口。块 RAM 接口可实现为每个时钟周期提供两个样本的双端口接口。另一种接口类型只能支持每时钟周期一个样本。这样就会产生瓶颈。

HLS 优化方法的步骤 3 是让您克服这一瓶颈。为了在同一时钟周期内访问所有数据值，阵列必须被分区为单独的元，以创建 HEIGHT*WIDTH 端口 （如果每个端口都是双端口块 RAM 接口，那就是该数量的一半），从而让所有端口都能在同一时钟周期中读取。输出端口的情况类似。

注释：如需了解更多有关优化方法的信息，请参阅"优化方法"。

这将是一种高度并行化的设计，但也是规模很大的设计。

Loop1 层面：Loop1 进程中的逻辑用于处理二维矩阵中整个一行。把 PIPELINE 指令布置在此处可创建出每个时钟周期寻求处理一行的设计。同时这样也会展开位于其下的循环，创建更多逻辑。

这是一种大规模并行设计，运行速度比第一种慢但规模更小。

Loop2 层面：循环中的逻辑会尽量处理阵列中的一个样本。如果设计目的是每时钟周期处理一个样本，那么这就是需要流水线化的层面。

这样会导致 Loop3 完全展开。但由于 Loop2 每个时钟周期处理一个样本，这就成为了要求，而且通常是必要条件。在典型设计中， Loop3 中的逻辑一般是移位寄存器或字内部的处理位。要做每个时钟周期执行一个样本，您应让这些进程并行开展，故需要展开循环。

该设计每个时钟周期处理一个数据样本并且只在需要的地方创建并行逻辑，从而实现这一规模的数据吞吐量水平。

Loop3 层面：如上文所述，在这种情况下 Loop3 中的逻辑一般会负责位级任务或数据移位任务。这个层面高于运行在每个数据样本上的层面。例如，如果 Loop3 包含一个移位寄存器操作且 Loop3 实现了流水线化，就会告知工具每个时钟周期移位一个数据值。设计只会返回到 Loop2 中的逻辑且在所有样本移位完成后读取下一输入。

本例中理想的流水线化的位置就是 Loop2。

在处理基于帧的代码时，您应考虑在循环层面进行流水线化，一般流水线化运行在样本层面的循环。如果存疑，在 C 语言代码中布置一条打印命令，使用 C 语言仿真来确认这是您希望在每个时钟周期上执行的层面。

根据上文的详细说明，通常在基于帧的设计中使用 ARRAY_PARTITION 指令将阵列分区为较小的块 （或对位于接口上的阵列来说，多个端口），可起到消除性能瓶颈的作用。

基于样本的 C 语言代码

基于样本的 C 语言代码的概要示例见下。这种编码方式的主要特征是函数在每个传输事务过程中处理一个数据样本。

```
void foo (data_t *in, data_t *out) {

  static data_t acc;

  Loop1: for (int i=N-1;i>=0;i--) {
  acc+= ..some calculation..;
  }

  *out=acc>>N;
}
```

在基于样本的函数中，数据以 scalar、指针或 `hls::stream` 变量形式提供。

指针或 `hls::stream` 在函数内可多次访问，但在基于样本的描述中它们只能访问一次。

基于样本的编码方式的又一特点是函数往往内含静态变量。这一变量的值必须在函数调用之间存储起来，例如利用累加器或样本计数器。

要实现 II=1，即每个时钟周期读取一个数据值，该函数必须流水线化。这样做会展开一切循环，创建额外的逻辑，但这是不可避免的。如果 Loop1 实现了流水线化，那就需要至少 N 个时钟周期才能完成。只有到那时该函数才能读取下一个 x 输入值。

在处理运行在样本层面的 C 语言代码时，最佳策略往往是把函数流水线化。因为基于样本的设计中的循环一般运行在执行移位寄存功能的阵列上，把这些阵列分区为独立元以确保所有样本在一个时钟周期内完成移位的做法比较常见。否则移位操作会被限定为读取来自或写入样本到双端口块 RAM。

这里的解决方法是流水线化函数 foo。这样做即可得到每个时钟周期处理一个样本的设计。

RTL 验证

Vivado HLS 内部的 RTL 验证流程是完全自动运行的。在 RTL/C 语言协同仿真中， C 语言仿真使用的 C 语言测试平台会被重复使用，综合后的函数则由 RTL 设计代替。使用正确的接口协议对进出 RTL 设计的数据排序则是由 Vivado HLS 自动完成。

因为重用了 C 测试平台，所以不需要创建 RTL 测试平台。

有部分设计选项可能会妨碍 RTL/C 语言协同仿真。要执行 RTL/C 协同仿真，必须符合以下条件。

- 顶层函数必须使用 ap_ctrl_hs 或 ap_ctrl_chain 块级接口综合。
- 或者设计必须是纯合并性设计。
- 或顶层函数的初始化时间间隔必须为 1。
- 或接口必须全部是使用 ap_fifo、 ap_hs 或 axis 接口模式传输和实现的阵列。

如果这些条件之一未能满足， C/RTL 联合仿真会中止并提示下列消息：

```
@E [SIM-345] Cosim only supports the following 'ap_ctrl_none' designs: (1)
combinational designs; (2) pipelined design with task interval of 1; (3) designs with
array streaming or hls_stream ports.
@E [SIM-4] *** C/RTL co-simulation finished: FAIL ***
```

IP 封装

在设计完成之后，会使用 Vivado HLS 的导出 RTL 功能创建一个适用于 IP 目录的 IP 封装。对包含 AXI4-Lite 接口的设计，该 IP 封装内含有用于接口编程的必要软件驱动程序文件。

Vivado HLS 提供多种封装选项。要使用高生产力 IP 集成器方法，应使用 IP 目录格式，同时接口应是 AXI 接口。

设计分析与优化

任何设计方法的必备一环是使用高生产力的设计分析与改进流程。《Vivado Design Suite 用户指南：高层次综合》(UG902) [参照 2]中的链接描述了使用 Vivado HLS 进行 C 语言仿真、C 语言调试、综合、分析、RTL 验证以及 IP 封装的流程。

创建设计和提升设计性能的流程可总结为：

- 仿真 C 语言代码，验证设计的正确性。

- 综合初始设计。
- 分析设计性能。
- 创建新解决方案并添加优化指令。
- 分析新解决方案的性能。
- 继续创建新解决方案和优化指令，直至要求得到满足。
- 验证 RTL 的正确性。
- 把设计封装为 IP 并包含到您的系统中。

生产效率最高的方法是使用 C 语言仿真同时验证设计并在综合前确认结果的正确性。 C 语言仿真速度带来的好处是高层次设计流程的主要优势所在。相对于调试因使用错误规格造成的性能问题，确认 C 语言设计的正确性才是更具备生产力成效的做法。

确保报告的实用性

在实现初步综合结果后，第一步是审核结果。如果综合报告内含任何未知值 （显示为问号“？ ”），就必须加以解决。为判断优化指令是否能提升设计性能，关键在于比较解决方案：在比较中时延必须是已知值。

如果循环有变量绑定， Vivado HLS 就不能判断完成循环的迭代数量。由于变量绑定的存在，即便循环的一次迭代的时延是已知的， Vivado HLS 也不能判断完成循环的全部迭代的时延。

审核设计中的循环。在综合报告中，审核“Latency > Details > Loops”部分中的循环。从循环层级中报告未知时延的最低层循环开始，因为这一未知性会沿层级向上传输。变量绑定循环可能位于层级中的较低层。审核综合报告的“Latency > Details > Instance”部分，查看是否有子函数显示为未知值。打开任何显示未知时延值的函数的报告，重复这一流程，直至找出变量绑定循环。

除了使用综合报告，还可以使用分析视角。

在找到变量绑定循环后，添加 LOOP_TRIPCOUNT 指令以指定该循环的迭代计数，或使用 C 语言代码的激活 (assertion) 来设定限值。如需了解更多信息，请参阅 《Vivado Design Suite 用户指南：高层次综合》(UG902) [参照 2] 中的链接。

如果使用 LOOP_TRIPCOUNT 指令，可考虑将该指令以 `pragma` 的形式添加到源代码中，因为该指令在每个解决方案中都需要。

如果您清楚存在有变量绑定的其他循环，为这些循环设定迭代限值。否则就要重复综合，使用同样的自下而上方法，直至顶层报告包含真实的数值。

设计分析

可以使用三种不同的技巧开展设计分析。

- 综合报告。
- 分析视角。
- RTL 仿真波形。

提示：在分析结果之前，应审核控制台窗口或日志文件，查看已经完成的、跳过的或失败的优化。

综合报告和分析视角可用于分析时延、时间间隔和资源估算。如果现在存在一个以上的解决方案，使用 GUI 中的报告按钮并列比较这些解决方案。该功能和分析视角一样，只提供在 GUI 中。不过值得注意的是使用 `vivado_hls -p project_name` 可以在 GUI 中打开使用批处理模式创建的项目加以分析。

此时层级方法一样有用。从顶层开始，判断哪些任务对时延、时间间隔或面积影响最大，然后深入核查这些任务。反复沿层级向下，直至找到您认为能够或应该对实现您的目标发挥更理想性能的循环或函数。在改进这些函数或循环后，产生的改进效果会沿层级向上传递。

与综合报告相比，使用分析视角可以更加方便地沿设计层上下移植。除此之外，分析视角还能够针对与 C 语言代码交叉关联的受调度操作和资源使用提供详细视图。

在深入探查细节之前，先使用分析视角的详细调度视图查看宏层面行为，这能提供一定帮助。这些操作一般按 C 语言代码的执行次序列出。需要牢记的是， Vivado HLS 会试图把一切调度到时钟周期 1 中，并争取在 1 个时钟周期内完成。

- 如果看到操作呈现从左上到右下的总体偏离，可能是数据关联性或执行代码中固有的任务造成的。每个操作都需要前一个操作完成后才能开始。
- 如果看到操作调度为顺序执行，但突然发生大量项目同步执行或与此相反的情况，可能说明有瓶颈存在 （比如 I/O 端口或 RAM 端口）。这时设计必须反复等待，随后一切进入并行执行。

除了综合报告和分析视角，也可使用 RTL 仿真波形帮助分析设计。在 RTL 验证过程中可以保存走线文件并使用合适的查看器查看。请参阅 《Vivado Design Suite 教程：高层次综合》(UG871) [参照 3] 中的 RTL 教程部分详细了解。此外，导出 IP 封装并在 `project_name/solution_name/impl/ip/verilog or vhdl` 文件夹中打开该 Vivado RTL 项目。如果已经执行了 C/RTL 协同仿真，该项目会包含一个 RTL 测试平台。

使用该 RTL 用于设计分析时务必十分小心谨慎。如果修改 C 语言代码或添加优化指令，在综合重新执行后，很可能得到名称迥异的显著不同的 RTL 设计。每次有新设计生成都需要花时间重复了解 RTL 细节，而且会使用显著不同的名称和结构。

总之，应沿层级向下找出需要进一步优化的任务。

设计优化

在开展任何优化之前，建议在项目中创建新的解决方案。使用解决方案可以比较不同的结果集合。不仅可以比较结果，还可以比较日志文件乃至输出 RTL 文件。

高性能设计的基本优化策略是：

- 创建初始或基准设计。
- 流水线化循环和函数。
- 解决限制流水线化的任何问题，例如阵列瓶颈和循环相依性 （使用 ARRAY_PARTITION 和 DEPENDENCE 指令）。
- 应用 DATAFLOW 优化以同时执行循环和函数。
- 有时可能需要对代码进行调整以满足性能要求。
- 缩小数据流存储器通道的大小并使用 ALLOCATION 和 RESOUCE S 指令进一步缩小面积。

总之，目标是永远以满足性能要求优先，其次是缩小面积。如果策略的目的是利用更少的资源创建设计，只需要忽略用于提升性能的步骤即可。

在整个优化流程中，强烈建议在综合后审核控制台输出 （或日志文件）。如果 Vivado HLS 无法实现某优化的设定性能目标，它会自动宽松目标 （时钟频率除外）并以能够满足的目标创建设计。应审核综合的输出以掌握已经完成哪些优化。

如欲了解有关应用优化的具体详情，请参阅 《Vivado Design Suite 用户指南：高层次优化》(UG902) [参照 2]。

XILINX

第 5 章

系统集成

简介

和 shell 设计开发相同，在流程中利用 Vivado® IP 目录和 IP 集成器是高效系统集成方法的关键。如果遵循高层次设计方法的早期步骤，那么该阶段的设计进程包含以下几项：

- 一个预先验证的 shell。电路板级接口已经开发完成并验证成功。
- 作为设计的核心功能，将一系列经预先验证的 IP 块封装起来用于 Vivado IP 目录。
- 用于 IP 级接口的 AXI 接口，有助于 IP 集成器的设计辅助功能来进行自动化设计创作。
- 已创建好一个用于验证平台的系统级测试 shell。

并行开发和验证的系统组件现已准备就绪，可用于系统集成。

完成初始系统集成后，您现在便拥有了整个流程自动化所需的一切，可以轻松地生成更多的新设计。

初始系统集成

设计集成进程可概括如下：

1. 基于 shell 设计创建一个新的 Vivado 项目
2. 添加所有 IP 块，并使用 IP 集成器将 IP 连接到起来
3. 验证系统并通过实现 FPGA 比特流处理设计。

系统集成项目

使用其中一个参考 shell 设计创建一个新的系统设计集成项目，如下所示：

1. 打开 Vivado 项目的 shell 设计并选择“File > Save Project As”在新项目中保存 shell 设计。
2. 创建一个新的 Vivado RTL 项目 （无 RTL 源）和相同的目标器件或目标板。然后选择“Create Block Design”，并在控制台中将使用 `write_bd_tcl` 保存的 Tcl 脚本做为源，在新项目中重新生成 shell 块设计。
3. 将所有的核设计 IP 块添加到项目 IP 资源库。

自动初始系统

IP 集成器可提供设计辅助功能，有助于集成系统。建议打开 shell 设计并将所有的 IP 块添加到菜单中。如果在所有的 IP 和 shell 接口中都使用 AXI 接口，设计辅助功能将被激活，并会提供连接建议。

设计助理确认哪些连接是合法的。除了简单地自动化连接之外，它还可以自动添加任何所需的 AXI 互连逻辑，例如将 AXI4-Stream 接口连接到 AXI 内存映射端口。如需了解设计辅助功能的完整介绍，请参阅 《Vivado Design Suite 用户指南：采用 IP 集成器设计 IP 子系统》(UG994) [参照 8] 中的链接。

培训：如果在设计中使用了多个时钟域，请参阅 Vivado Design Suite QuickTake 视频：在 Vivado IP 集成器中使用多个时钟域。

通过实现设计辅助功能中所不支持的连接，如标量信号以及任何非 AXI 总线接口，来完成块设计。

最后，使用验证设计功能来确保设计没有违反设计规则。如需了解使用存储器映射接口或处理器的设计，请参阅 《Vivado Design Suite 用户指南：采用 IP 集成器设计 IP 子系统》(UG994) [参照 8] 中的链接。

当设计完成并成功验证后，赛灵思建议您使用 `write_bd_tcl` 命令。`write_bd_tcl` 命令可以在 Tcl 文件中保存重新生成整个系统所需的所有内容。

系统验证与实现

在 IP 集成器中完成系统块设计后，您可以通过生成输出产品和为设计创建一个顶层 HDL 封装程序来开展整个系统的验证和实现。如欲了解上述使用 IP 集成器方式的完整流程，请参阅 《Vivado Design Suite 教程：高层次综合》(UG871) [参照 3] 中的下述章节：

- 第 9 章“在 IP 集成器中使用 HLS IP”
- 第 10 章“在 Zynq SoC 设计中使用 HLS IP”

用于 shell 验证的同一个 RTL 测试 shell 也用于执行系统级验证。IP 块和 shell 设计等各个部分已经实现预先独立验证。现在的任务就是验证整个系统。在初始阶段，请您通过下列方式重点确认系统级连接：

1. 确保 shell 为处理流水线中第一个块的输入提供正确数据。
2. 确保第一个块向下一块提供正确的输出。

注释：使用最小的数据量，确保最高速的系统级仿真运行。这时请您先集中精力确认块连接。

验证了顶层的连接后，便可以执行完整详尽的系统仿真。

重要提示：如下面“自动初始系统”所示，针对任何未能通过系统级验证的 IP，您可采用这种先进的方法迅速开展重新设计，然后通过高度脚本化的方式快速重新生成整个系统。

在系统充分验证后，设计便可以用比特流的方式实现。如果使用 Vivado HLS 通过 C/C++创建了大部分设计 IP，那么 RTL 已经自动完成定时以确保 RTL 综合时时序准确。

注释：经过 RTL 综合后，如果 IP 块之间存在不符合时序的时序路径，应考虑在 HLS 过程中使用 `INTERFACE` 指令来注册接口端口。

为了提升运行时间，可以考虑在生成输出结果时使用“Out of Context per IP”模式。这一选项可生成一个经综合的输出产品，并将其缓存，并且只在 IP 块发生改变的情况下重新执行综合，从而降低系统的实现时间。

自动初始系统

虽然只在系统集成阶段才执行完整的系统级验证，这一点或许令人担忧，但实际上它正是这种方法的优势之一。一次完整的 RTL 系统级仿真可能耗时较长，在项目开发中执行多次这样的仿真占用了大量项目开发时间，这通常是开发过程中最耗时的任务。

本方法重点关注：

- 并行开发
- 用 C/C++ 仿真的方式来验证 IP 可将验证速度提升几个数量级。
- 创建与验证块级 IP
- 现有与已验证 IP 的重用

由于 Vivado Design Suite 的高度自动化，使得这一方法卓有成效。本节内容展示了如何使用 Tcl 脚本快速轻松地重新创建整个系统，即使系统集成过程中发现了问题也不会有影响。

Vivado 项目自动化

在 Vivado IDE 中执行的所有操作均以 Tcl 命令的形式采集到项目日志文件中。在批处理模式下，这些命令允许您重复所有的动作，显著减少了执行任务所需时间。在此方法中，这种自动生成 Tcl 命令的方式可以将下列任务实现高度自动化：

- 创建项目
- 在项目中添加 IP
- 系统仿真
- 系统实现

下面的代码示例展示了您可以轻松地针对项目创建、在项目中添加 IP 资源库、创建块设计、将项目处理成比特流等任务实现流程的完全自动化。

```
# Set project parameters
set my_part xc7z020clg484-1
set my_board_part xilinx.com:zc702:part0:1.0

# Set the paths to auto-adjust to the local directory
# Define project and IP repository locations
set my_files [pwd]
set projdir $my_files/project_1
set repo_dir $my_files/../my_ip/ip
puts "Using project directory $projdir"
puts "Using repository directory $repo_dir"

# Create the Project
set projname project_1
create_project -force $projname $projdir -part $my_part
set_property board_part $my_board_part [current_project]

# Create IP repository
set_property ip_repo_paths $repo_dir [current_fileset]
update_ip_catalog -rebuild

# Create the block design
source ./design_IPI.tcl
```

```
# Create output products and HDL wrapper
generate_target all [get_files
$projdir/$projname.srcs/sources_1/bd/$design_name/$design_name.bd]
make_wrapper -files [get_files
$projdir/$projname.srcs/sources_1/bd/$design_name/$design_name.bd] -top
add_files -norecurse
$projdir/$projname.srcs/sources_1/bd/$design_name/hdl/${design_name}_wrapper.v
update_compile_order -fileset sources_1
update_compile_order -fileset sim_1

# Implement the bitstream
launch_runs impl_1 -to_step write_bitstream
wait_on_run impl_1
```

执行上述操作的 Tcl 命令直接从项目日志文件复制得来。或者，您可以使用“Save > Write Project Tcl”在 Vivado 中轻松将流程校本化并实现 shell 创建、 shell 验证和系统集成的流程自动化。

Vivado HLS 自动化

Vivado HLS 使用 IDE 为每个项目创建一个 Tcl 文件。下面的代码示例展示了在 C 语言设计流程中所有步骤的 Tcl 命令：仿真 C 语言代码、使用优化指令综合 C 语言代码、验证 RTL 是否正确，并创建一个 IP 封装，以及确认 RTL 综合符合时序。

```
# Create a project and add files
open_project proj_matrixmul
set_top DESIGN_TOP
add_files matrixmul.cpp
add_files -tb matrixmul_tb.cpp

# Create a solution
open_solution "fast"
set_part {xc7z020clg484-1}
create_clock -period 4 -name default

# Add optimization directives
set_directive_pipeline "cholesky/"
set_directive_array_reshape -type complete -dim 2 "matrixmul" a
set_directive_array_reshape -type complete -dim 1 "matrixmul" b

# Simulate, Synthesize, Verify and package outputs
csim_design
csynth_design
cosim_design
export_design -format ip_catalog -evaluate verilog
```

可对这种自动生成的 Tcl 的文件进行编辑，为 C 语言 IP 开发流程的任何部分实现自动化。例如，可以创建只用来执行 C 语言仿真的脚本。验证设计后，像上述这样的完整脚本可用于将设计综合成为一个封装 IP。

IP 集成器自动化

IP 集成器的 `write_bd_tcl` 命令不仅保存了一个 Tcl 脚本以重现您的操作，同时也会优化脚本，使其专门用于创建最终块设计。只要执行该脚本就会重新创建块设计。由于使用了 IP 资源库中的 IP 重新创建块设计，所以如果 IP 更新过，也将使用最新的 IP 重建设计。这种自动化水平使您能够快速重新创建块设计：

- shell 设计可以在一个新的设计项目中重新生成，执行修改，并轻松地生成一个新的 shell。
- shell 设计可以在一个新的验证项目中重新生成，并且 IP 验证可以方便地添加到设计中。
- shell 设计可以在系统集成项目中重新生成，并且核设计 IP 可集成到系统中。

整个方法中的每个步骤都可以以高效又生产力显著的方式重新执行。

系统整体自动化

通过使用 Makefile 执行的脚本能够进一步提供生产力优势。 Makefile 能定义一组相关性指令。例如，下列任务必须依序执行：

- 任务 A：在 C 语言中仿真一个 IP。
- 任务 B：在 IP 目录中综合 IP。
- 任务 C：将 IP 在系统级集成。

当用 Makefile 来执行任务 C 时，它会进行自动检查，以确定任务 B 的输出是否存在。如果输出不存在，它将试图执行任务 B，并检查任务 A 的输出是否存在，以此类推。

使用 Makefile 来执行 Tcl 脚本同上述流程类似，因此，可以直接更新任何 IP 或 shell 设计，并发出一个命令来重新创建整个系统，如果出现任何如下情况，执行将停止：

- 需要检查 C 语言仿真的结果
- 因添加验证 IP 而进行重新创建 shell 设计
- 如需重新综合一个 IP，先使用 RTL 仿真来验证 IP，然后重新构建系统。
- 完成 FPGA 变成后。

由于高度自动化，此种方法涉及所有一切将链接成为一个生产力显著的流程，这也导致该方法的部分流程可以并行执推进，并等到系统集成后执行系统级仿真。在系统的第一版本创建完成后，整代系统就实现了完全自动自动化。

设计未来

使用高层次生产力设计方法的最后一个好处是可以非常轻松地基于初始设计来创建衍生设计。这一生产力提升的两大使能因素是：

- 用 C 语言开发 IP
- 采用自动化实现流程

用 C 语言开发 IP

除了本指南中列出的部分之外，另外一个使用 C 语言开发 IP 优势，即轻松重新定位涉及的方式是：利用同一个源创建衍生设计。

上文 Vivado HLS 脚本示例中，采用了下列 Tcl 命令来定位 250 MHz 时钟的 Zynq®-7000 SoC 器件：

```
set_part {xc7z020clg484-1}
create_clock -period 4 -name default
```

如果要创建准确定时的 IP 块，例如，在一个 300 MHz 的 Kintex® UltraScale™ 器件中实施，只需要更新优化约束就能够实现：

```
set_part {xcku025-ffva1156-2-i}
create_clock -period 300MHz -name default
```

此外无需任何改动。在目标技术中， Vivado HLS 只需要创建实现特定频率的设计。在更高速的技术里，该设计或许用更少的时钟周期就能完成 （相反地，在更慢速的技术中需要更多的时钟周期），但无需重新编码或重新优化。

使用 C 语言代码创建的内容越多，设计也就更容易与新技术和/或时钟频率匹配。原有 RTL 块可能仍需要被重新实现，以适应不同的时序参数。

自动化实现流程

如果在一个完全校本化的流程中实现 FPGA，您就可以通过参数化脚本创建衍生设计来进一步提高设计复用率与生产力。

上述脚本示例中使用了下列代码。此示例使用了 Vivado 脚本，它也同样适用于强化 Vivado HLS 和 IP 集成器脚本。

```
# Set project parameters
set my_part xc7z020clg484-1
set my_board_part xilinx.com:zc702:part0:1.0
```

如需仅用一个顶层项目参数脚本设置项目中所有内容，请变更之前的代码以匹配下列内容：

```
# source project-level parameters
source project_top.tcl
# Set project parameters
set my_part $target_device
set my_board_part $target_board
```

本示例中 project_top.tcl 的内容为：

```
set target_device xc7z020clg484-1
set target_board xilinx.com:zc702:part0:1.0
```

修改这一脚本会使项目以高度自动化的方式进行重新定位并重新实现。

其他相关注意事项：

- 如果您遵循 C 测试平台的有关建议， C 语言仿真将实现自动检查。
- Vivado HLS 基于新参数生成新的 IP。
- 由 Vivado HLS 创建的 RTL 通过 RTL 仿真自动验证。
- Vivado 项目生成脚本将基于更新参数创建一个新项目。
- IP 集成器脚本像之前一样执行 IP 连接，然后设计自动化功能将使用最合适的连接。
- 您可以在系统集成这一步暂停流程，然后修改设计。
- 使用更新目标器件和时钟频率可将完成后的系统实现为比特流。

强化您的 Tcl 脚本以促进一定程度的参数化将大大提高您的能力，更加贴近以交钥匙方式创建衍生设计的流程。

XILINX

附录 A

附加资源与法律提示

赛灵思资源

如需了解答复记录、技术文档、下载以及论坛等支持性资源，请参阅赛灵思技术支持。

解决方案中心

如需了解设计周期各阶段有关器件、软件工具和 IP 等的技术支持，请参阅赛灵思解决方案中心。相关专题包括设计辅助、建议和故障排除提示等。

Documentation Navigator 与设计中心

赛灵思 Documentation Navigator (DocNav) 提供了访问赛灵思文档、视频和支持资源的渠道，您可以在其中筛选搜索信息。打开 DocNav 的方法：

- 在 Vivado IDE 中，单击“Help > Documentation and Tutorials”。
- 在 Windows 中，单击“Start > All Programs > Xilinx Design Tools > DocNav”。
- 在 Linux 命令提示中输入 `docnav`。

赛灵思设计中心 （Xilinx Design Hubs）提供了根据设计任务和其他话题整理的文档链接，您可以使用链接了解关键概念以及常见问题解答。访问设计中心：

- 在 DocNav 中，单击“Design Hubs View”视图。
- 在赛灵思网站上，查看设计中心页面。

参考资料

1. 《采用 Vivado® 高层次综合开展 FPGA 设计介绍》(UG998)
2. 《Vivado Design Suite 用户指南：高层次综合》(UG902)
3. 《Vivado Design Suite 教程：高层次综合》(UG871)
4. 《Vivado Design Suite 用户指南：版本说明、安装和许可》(UG973)
5. 《Vivado Design Suite 教程：创建和封装定制 IP》(UG1119)

6. 《Vivado Design Suite 用户指南：采用 System Generator 开展基于模型的 DSP 设计》(UG897)
7. 《UltraFAST 设计方法指南 （适用于 Vivado Design Suite）》(UG949)
8. 《Vivado Design Suite 用户指南：采用 IP 集成器设计 IP 子系统》(UG994)
9. 《Vivado Design Suite 用户指南：系统级设计输入》(UG895)
10. 《使用 Vivado IP 集成器集成基于 AXI4 的 IP》(XAPP1204)
11. Vivado Design Suite 技术文档
12. 赛灵思 IP 页面

培训资料

赛灵思提供多种多样的培训课程和 QuickTake 视频，可帮助用户进一步了解有关本文档中提出的概念。使用以下链接获取相关培训资料：

1. 基于 C 语言的设计：采用 Vivado HLS 工具开展高层次综合培训课程
2. 基于 C 语言的 HLS 编码硬件设计人员培训课程
3. 基于 C 语言的 HLS 编码软件设计人员培训课程
4. Vivado Design Suite QuickTake 视频教程
5. Vivado Design Suite QuickTake 视频：高层次综合
6. Vivado Design Suite QuickTake 视频：Vivado 高层次综合入门
7. Vivado Design Suite QuickTake 视频：验证 Vivado HLS 设计
8. Vivado Design Suite QuickTake 视频：创建不同类型的项目

请阅读：重要法律提示